实战深度学习算法：零起点通关神经网络模型

（基于 Python 和 NumPy 实现）

徐彬 著

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

深度学习是机器学习的重要分支。本书系统地介绍了如何用 Python 和 NumPy 一步步地实现深度学习的基础模型，无须借助 TensorFlow、PyTorch 等深度学习框架，帮助读者更好地理解底层算法的脉络，进而进行模型的定制、优化和改进。全书由简到难地讲述感知机模型、多分类神经网络、深层全连接网络、卷积神经网络、批量规范化方法、循环神经网络、长短时记忆网络、双向结构的 BiGRU 模型等神经网络模型的必要算法推导、实现及其实例，读者可直接动手调试和观察整个训练过程，进一步理解模型及其算法原理。

本书适合没有深度学习基础，希望进入此领域的在校学生、研究者阅读，也适合有一定基础但不满足于“调包”和“调参”的工程师学习，还可供想要深入了解底层算法的研究人员参考阅读。

图书在版编目（CIP）数据

实战深度学习算法：零起点通关神经网络模型：基于 Python 和 NumPy 实现 / 徐彬著.
北京：电子工业出版社，2019.9
ISBN 987-7-121-37171-4
I. ①实… II. ①徐… III. ①机器学习—算法—研究 IV. ①TP181
中国版本图书馆 CIP 数据核字（2019）第 158449 号

责任编辑：孙学瑛 sxy@phei.com.cn
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：720×1000　1/16　　印张：14　字数：280 千字　彩插：2
版　　次：2019 年 9 月第 1 版
印　　次：2019 年 9 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

前言

以深度学习为代表的人工智能技术深刻地影响着我们的生活方式，从图像识别、语音识别、机器翻译到智能医诊、自动驾驶、智能风控 …… 在多个应用领域不断地刷新纪录。

深度学习近年来之所以能取得颠覆性突破，一方面，归功于“数字化”对社会的渗透使得大量数据得以积累；另一方面，受益于单位成本下硬件算力的提升，推动了复杂模型的商用；然而最根本的，还是来自深度学习背后基础算法的巧思妙想与厚积薄发。

只有深入了解深度学习的算法原理，才能更灵活、高效地运用于实践当中。现有的深度学习框架将算法使用简化为“调包”和“调参”，降低了使用成本，然而却没有降低学习成本。对于算法，最有效的学习方式是理解原理并动手实践。从原始论文可以查阅算法的详解和推导，却不容易复现结果。主流的深度学习框架多采用计算图模型，不容易调试或观察，对希望深入了解算法的初学者并不友好。致力于用深度学习方法创造社会价值的从业者，也需要看清底层算法的脉络，来做模型的定制、优化和改进。

内容组织逻辑

本书的特点是“原理 + 实践”。按照“**带着问题看算法**”的逻辑来组织内容，所描述的每一种深度学习算法都围绕一个实际的目标问题展开，提供了基础算法的必要推导和实例，方便直观理解。

1. 提出问题。
2. 以问题为动机，引出模型。
3. 介绍模型原理、必要推导和实例。

4. 实现模型算法。

5. 解决问题与验证。

第 1 章至第 3 章，从感知机模型开始，步步渐进，介绍多分类神经网络、深层全连接神经网络；第 4 章至第 6 章，描述卷积神经网络（CNN）的核心算法、学习策略优化方法，以及深度学习的批量规范化方法；第 7 章至第 9 章，系统介绍了序列模型，基础的循环神经网络（Vanilla RNN）、长短时记忆网络（LSTM）和双向结构的 BiGRU 模型，以及序列模型适用的正则化方法。每章均以真实数据集作为目标问题，引出算法原理，**不借助深度学习框架**给予实现，最后完成数据集的验证。

阅读和使用

对于初学者，在阅读本书前，不需要具备机器学习基础，可以通过案例和模型概述等章节入门深度学习的概念；如果会使用 Python 语言简单编程，还可以结合书中的案例，动手了解各种模型所能解决的问题。

对于已有深度学习框架使用经验、希望了解底层算法的读者，可以查阅重要算法的原理、前向计算和反向传播的推导步骤，并在各章节的算法实现部分了解全部算法的实现过程。

附录部分包含了书中讨论的深度学习算法所涉及的数学基础，方便初学者速查和理解其直观意义。如需概念的严格定义和展开论证，可参考相关教材和专著。

通过阅读本书，希望读者可以：

- 理解深度学习主要的核心模型。
- 灵活复现重要论文、验证新方法。
- 自由替换模型中的底层算法，取得一手实验结果。
- 针对自己工作的特定场景，对算法做定制和优化，实现工程应用。

书中提供的算法实例基于 Python 和 NumPy 库实现，可以在 GitHub 公开代码库中获取`https://github.com/AskyJx/xDeepLearningBook`。

致谢

感谢为此书提供支持和做出过贡献的每个人。感谢家人的支持，他们是我动力的源泉。重庆大学郭平教授拨冗审阅了部分内容，提出了宝贵意见，对本书质量的提高有很大帮助，在此向他表示衷心的感谢。感谢一起工作过的同事和旅途巧遇的伙伴给予的启发，打开了前沿技术新世界的大门。电子工业出版社的孙学瑛女士在本书写作和出版过程中给予了很多帮助，在此特向她致谢。

由于作者水平有限，书中难免存在错误和不足，敬请专家和读者批评指正。如有意见或反馈，可以发邮件至xu.bin.sh@foxmail.com。

也欢迎在知乎上交流https://www.zhihu.com/people/xu-jerry-82。

徐彬

2019 年 7 月

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在 下载资源 处下载。
- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/37171

目录

符号表

a, x, w	标量
$\boldsymbol{a}, \boldsymbol{x}, \boldsymbol{w}$	n 阶张量（$n \geqslant 1$）
$\vec{a}, \vec{x}, \vec{w}$	向量（一阶张量）
$\boldsymbol{M}$	矩阵（二阶张量）
$\boldsymbol{x}^{\mathrm{T}}, \boldsymbol{M}^{\mathrm{T}}$	向量 $\boldsymbol{x}$ 和矩阵 $\boldsymbol{M}$ 的转置
$f :\to \mathbb{R}$	输出实数标量值的函数
$\boldsymbol{f} :\to \mathbb{R}^m$	输出 m 维实数向量的函数
$\mathrm{diag}(f)$	函数 f 的值构成的对角方阵
I	单位阵
$f'(x)$	函数 f 的一阶导函数
$\cdot$	点积运算符
$\oplus$	矩阵对应位置元素相加运算符
$\otimes$	矩阵对应位置元素相乘运算符,用于 Hadamard 乘积运算
$\odot$	互相关运算符，用于神经网络中的卷积运算
$\circledcirc$	经典卷积运算符

符号表

$z^{(l)}$	神经网络第 l 层原始输出
$a^{(l)}$	神经网络第 l 层激活后输出
$\delta^{(l)}$	回传至神经网络第 l 层误差项

第 1 章

基础分类模型

> “By the study of systems such as the perceptron, it is hoped that those fundamental laws of organization which are common to all information handling systems, machines and men included,may eventually understood.”
>
> “探究感知机这类系统，我们有望**最终理解**那些**基本法则**，那些将**信息认知**赋能于机器和人类的基本法则。”
>
> Frank Rosenblatt@Connell Aeronautical Laboratory,1958

本章简要介绍深度学习的概念，接着提出第一个目标问题：二分类问题；然后介绍感知机模型的原理及其解决问题的过程；再给出感知机的一种算法实现供参考；最后在目标问题上，观察分类效果。

1.1 深度学习简介

机器学习是解决人工智能问题的方法之一，深度学习是机器学习方法下的一个重要子类别，三者之间是依次从属的关系：

$$\text{深度学习} \subset \text{机器学习} \subset \text{人工智能}$$

在图像和语音识别、机器翻译、策略游戏等领域，深度学习的表现超越了传统机器学习，深度学习模型的结构也较传统机器学习更加复杂。完整的深度学习模型往往是由不同处理层组合而成的复合结构模型，庞大而复杂；如果把整体模型分解开来，就会发现组成模型的要素是一个个设计精巧的基础算法，这些基础算法一部分来自积淀深厚的传统机器学习方法，一部分是深度学习的创新方法；在这些基础算法的支撑下，深度学习在诸多应用领域取得了前所未有的突破。

深度学习处理问题的过程可以描述为：通过在数据集上的迭代训练，捕捉分析对象的特征，并学习特征和输出表达之间的联系，继而根据同类对象的输入特征，实现对输出表达信息的推理。

通过接下来的第一个目标问题，可以直观理解这一过程。

1.2 目标问题：空间中的二分类

“能正确地提出问题，你已经解决了问题的一半。”

分类问题，是深度学习的常见问题场景，目的是把数据实例按照既定分类划分到不同类别归属上。二分类是最基本的分类问题。

想象 D 维空间里分布着 N 个线性可分的实例点 x_i，当 $D=3$ 时，这个空间即是一个便于理解的三维空间，其上的任意实例点 x_i，都有三个特征 $x_i^{(1)}$、$x_i^{(2)}$ 和 $x_i^{(3)}$；如果把实例点看成是 3 维实数向量 $\boldsymbol{x}=(x^{(1)},x^{(2)}\ x^{(3)})^{\mathrm{T}}$，则向量的三个特征分量决定了实例点 x_i 的二分类类别 $y_i=\{+1,-1\}$，如图1.1所示。二分类问题的目标，是找出实例点 x_i 的特征到输出类别 y_i 的对应关系。这个对应关系可以表达为映射函数 f：

$$y_i = f(x_i) \tag{1.1}$$

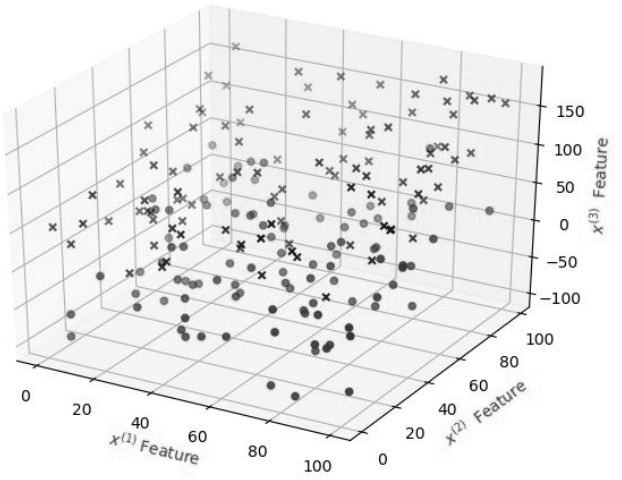

图 1.1 三维空间中的线性可分样本

如果能找到映射函数 f，就能根据实例点的特征，对空间中的样本做类别归属划分。感知机模型是找到映射函数的一种有效方法。

1.3 感知机模型

感知机（Perceptron）模型是一个简洁的分类模型，由 Rosenblatt 在 1957 年提出，能够对线性可分样本做二分类，是人工神经网络方法的理论基石。

1.3.1 感知机函数

感知机模型可以把目标问题中的从输入实例点 x_i 的特征到输出类别 y_i 的映射表示为如下函数:

$$f(x) = \text{sign}(\boldsymbol{w} \cdot \boldsymbol{x} + b) \tag{1.2}$$

其中“$\cdot$”表示两个向量的**内积（inner product）**运算；$\boldsymbol{w}$ 称为**权值向量（weight vector）**；b 是实数标量，称为**偏置（bias）**；$\text{sign}(\cdot)$ 是**符号函数**，将**自变量** $\boldsymbol{x}$ 进一步映射到 y_i 的输出类别 $\{+1, -1\}$ 上。

接下来定义空间里的超平面，用线性方程表示：

$$\boldsymbol{w} \cdot \boldsymbol{x} + b = 0 \tag{1.3}$$

这个超平面按照输出类别 $y_i = \{+1, -1\}$，将实例点 $\boldsymbol{x}$ 划分在平面两侧，如图1.2所示。

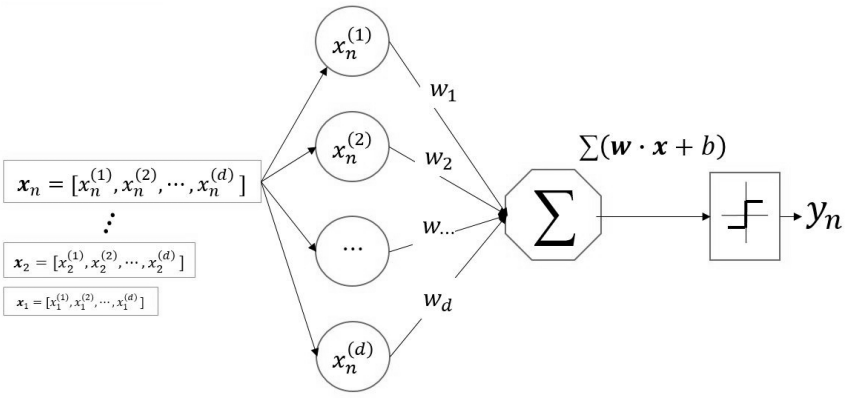

图 1.2　感知机模型：输入数据的每个样本都有 D 个特征，这些特征经过感知机函数处理，输出样本的二分类归属类别

对正实例点 $y_i = +1$ 有 $\boldsymbol{w} \cdot \boldsymbol{x} + b > 0$；

对负实例点 $y_i = -1$ 有 $\boldsymbol{w} \cdot \boldsymbol{x} + b < 0$。

由

平面上任意两个不同实例点，$\vec{x}_i$ 和 $\overrightarrow{x_{i+n}}$ 都满足

$$\vec{w} \cdot (\vec{x}_i - \overrightarrow{x_{i+n}}) = \vec{w} \cdot \overrightarrow{(x_{i+n} - x_i)} = 0$$

而内积的几何意义是向量 $\boldsymbol{a}$ 在另一个向量 $\boldsymbol{b}$ 方向上的投影长度与 $\boldsymbol{b}$ 长度的乘积，可知 $\vec{w}$ 是与分离超平面上任意切向量正交的法向量，这也是平面点法式方程的定义。

通过某种学习策略找到权值向量（法向量）$\vec{w}$ 和偏置（截距）b 这两个参数，确定划分正负实例点的超平面，就能对输入的 D 维空间上散布的线性可分实例点 x_i 做二分类预测。

1.3.2　损失函数

为了找到合适的权值向量 $\boldsymbol{w}$ 和偏置 b，先定义连续可导的损失函数（Loss Function），再将损失函数极小化，以找到所有可能的分离超平面中较优的一个。如果损失函数是凸函数，还可以用数值方法得到全局最优解，学习策略就

成为求解最优化问题。

$$\min_{f \in F} \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(x_i)) \tag{1.4}$$

能够完成样本分类的映射函数 f 可能不止一个，F 是所有能划分输入样本点的感知机模型 f 的集合；N 是训练样本容量；L 是模型 f 的损失函数。

损失函数仅仅是对一次预测的好坏度量，然而根据**大数定理**，当样本点容量 N 足够大时，样本点集合上，由损失函数得到的平均损失，趋近于总体分布在分类模型上的期望损失，从而可以用数理统计方法得到理想概率模型的近似解。

损失函数有多种经典选择，对二分类问题可以选择造成模型损失的误分类点到分离超平面的总距离来度量损失。

对任意一个样本点 $\boldsymbol{x}_i$，我们可以根据点到平面的距离公式，得出它到超平面的距离：

$$\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$$

$$\text{Distance} = \frac{|\boldsymbol{w} \cdot \boldsymbol{x}_i + b|}{\sqrt{\sum_{j=1}^{D} w_j^2}} \tag{1.5}$$

式中 $\sqrt{\sum_{j=1}^{D} w_j^2}$ 是法向量 $\boldsymbol{w}$ 到 D 维空间原点的欧氏距离，也称 $\boldsymbol{w}$ 的 L_2 范数（L_2 norm），记作 $\|\boldsymbol{w}\|_2$。

同时，根据超平面的定义，一个误分类的实例点 $\boldsymbol{x}_{\text{err}}$, 有:

$$-y_i(\boldsymbol{w} \cdot \boldsymbol{x}_{\text{err}} + b) > 0 \tag{1.6}$$

得到所有误分类实例点到超平面的总距离为：

$$-\frac{\sum y_i(\boldsymbol{w} \cdot \boldsymbol{x}_{\text{err}} + b)}{\sqrt{\sum_{j=1}^{D} w_j^2}} = -\frac{\sum y_i(\boldsymbol{w} \cdot \boldsymbol{x}_{\text{err}} + b)}{\|\boldsymbol{w}\|_2} \tag{1.7}$$

在优化损失函数使损失极小时，函数在同一权重参数处取得极值，所以函数取值的数值**缩放正倍数**不影响优化方法，进而损失函数可以进一步写为

$$L(\boldsymbol{w}, b) = -\sum y_i(\boldsymbol{w} \cdot \boldsymbol{x}_{\text{err}} + b) \tag{1.8}$$

直观理解，损失越小，误分类点距离超平面越近，直到所有样本点都被正确分类，损失降为 0，可知 $L(\boldsymbol{w}, b)$ 是 $\boldsymbol{w}$,b 的连续可导函数。

求解最优化问题：

$$\min_{f \in F} \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(x_i))$$

变成了损失函数极小时，参数 $\boldsymbol{w}$,b 的求解：

$$\min_{\boldsymbol{w}, b} L(\boldsymbol{w}, b) = -\sum y_i(\boldsymbol{w} \cdot \boldsymbol{x}_{\text{err}} + b) \tag{1.9}$$

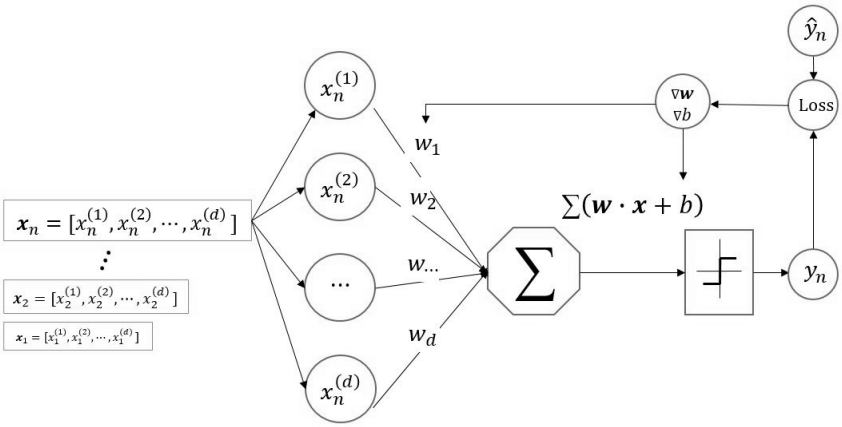

图 1.3　感知机的参数求解：通过误差损失，得到参数梯度，用梯度迭代地更新参数，以此找到一组可行解

1.3.3　感知机学习算法

一种求解算法是随机梯度下降（Stochastic Gradient Descent），先将 $\boldsymbol{w}$,b 的初始值设置为 0，然后用梯度下降法，让参数不断更新梯度 $\nabla \boldsymbol{w}$ 和 ∇b，以

极小化损失函数, 如图1.3所示。

$$\nabla_{\boldsymbol{w}}\text{Loss} = \frac{\partial[-\sum y_i(\boldsymbol{w}\cdot\boldsymbol{x}_{\text{err}}+b)]}{\partial \boldsymbol{w}} = -\sum y_i\boldsymbol{x}_{\text{err}} \tag{1.10}$$

$$\nabla_{b}\text{Loss} = \frac{\mathrm{d}[-\sum y_i(\boldsymbol{w}\cdot\boldsymbol{x}_{\text{err}}+b)]}{\mathrm{d}b} = -\sum y_i \tag{1.11}$$

为便于灵活调整每次梯度下降的尺度，引入参数“步长”或“学习率（Learning Rate）”超参数 η，根据随机选出的误分类点，来更新 $\boldsymbol{w}$,b 参数：

$$\begin{aligned}\boldsymbol{w} &= \boldsymbol{w} + \eta\boldsymbol{x}_{\text{err}}y_i \\ b &= b + \eta y_i\end{aligned} \tag{1.12}$$

可以证明，这个算法在线性可分数据集上是收敛的。

通过不断随机选取误分类点，更新 $\boldsymbol{w}$ 和 b，经过有限次迭代，能找到可以把线性可分正负实例点划分在两侧的一个分离超平面。参见图 1.4 空间中划分样本的分离平面。

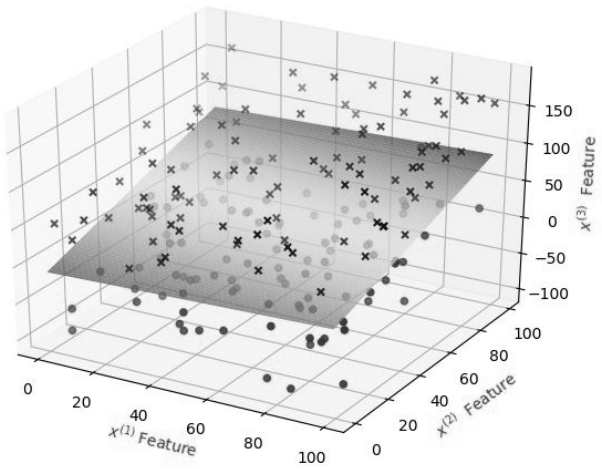

图 1.4 感知机模型训练得到的每一组参数，
定义了空间中划分样本的一个分离平面

1.4 算法实现

“纸上得来终觉浅，绝知此事要躬行。”

——[宋] 陆游《冬夜读书示子聿》

本书中讨论的算法，均提供了基于 Python 的一种实现，读者可以通过运行样例程序，观察数据处理过程，直观地了解算法原理。安装配置运行环境可参考下面的简要步骤。

1.4.1 环境搭建

本书中的算法，主要基于 Python 语言（不低于 Python 3）和 NumPy 库，使用类 UNIX 或 Windows 操作系统均可运行全部案例；算法的实现已做过优化，可以在普通配置的台式机或者笔记本电脑上运行，不需要 GPU 或其他的特别硬件支持。

安装 Python 编译器和科学计算库

Python 软件基金会网站（`http://www.Python.org/download`）提供了不同环境的编译器安装包，根据自己的操作系统环境，选择 Python 的安装文件，下载并配置 Python 运行环境。

建议使用 Anaconda 或者 Virtualenv，建立和管理多套虚拟 Python 环境。

本书样例需要安装 NumPy、SciPy 科学计算库和 matplotlib 图形库，不需要安装深度学习框架。

Python 环境配置完毕后，可以用以下命令安装所需的支持库。

```
1 python -m pip install numpy scipy matplotlib
```

集成开发环境

建议使用 PyCharm 编辑和运行示例源码、观察各个算法实例的执行结果，从 JetBrains 的网站（`http://www.jetbrains.com/pycharm`）可以下载 PyCharm 的社区版，这个版本支持各种主流操作系统且是免费的，可以满足本书全部实例的开发、调试需要。

也可使用 Jupyter Notebook 或其他自己熟悉的文本编辑器配置开发环境。

编译器安装和集成环境搭建不是本书的重点，这里不再展开介绍，参考各官网的指导步骤即可完成。

1.4.2 数据准备

为了生成空间里的 n 个线性可分实例点，先随机生成这些实例点的前两维坐标 $x^{(1)}$，$x^{(2)}$。

```
import numpy as np
n = 60
x1 = randrange(n, 0, 100)
x2 = randrange(n, 0, 100)
```

其中，自定义方法 randrange$(n, v_{\min}, v_{\max})$，用于产生 n 个取值范围在 $(v_{\min}, v_{\max})$ 之间的随机浮点数：

```
def randrange(n, vmin, vmax):
    return (vmax - vmin) * np.random.rand(n) + vmin
```

接下来定义一个平面，它由法向量 $\boldsymbol{w}\_$ 和偏置 $b\_$ 确定。

```
# 定义一个空间中的平面，用于生成样本点数据集
W_ = [2, -0.9, 1]
b_ = -1
```

三维空间中，落在这个平面上的点满足：

$$\sum_{i=1}^{3} w\_^{(i)} x^{(i)} + b\_ = 0$$

由此进一步推出随机实例点的第 3 维坐标值：

$$x^{(3)} = -\frac{w\_^{(1)} x^{(1)} + w\_^{(2)} x^{(2)} + b\_}{w\_^{(3)}}$$

```
x3 = -(w_[0] * x1 + w_[1] * x2 + b) / w_[2]
```

把第 3 维坐标值正向偏移一个随机量，这个原本落在平面上的点，即会位移到平面的上方。

用这个方法，得到 n 个正实例点。

```
x3_positive = x3 + randrange(n, 0, 100)
```

用同样的方法，在平面上生成另一组 m 个随机实例点，然后负向随机偏移，得到 m 个负实例点：

```
x3_negative = x3 - randrange(m, 0, 100)
```

对正负两类样本点设置正确的分类，用于训练模型。

```
# 正确分类 +1
y_[0:n] = 1
# 正确分类 -1
y_[n:n+m] = -1
```

用这种方法，以一个自定义的平面随机生成了空间中的样本点。

为了直观地了解模型效果，观察样本点，可以用绘图库 Matplotlib 和 mpl_toolkits 把这些样本点在空间中的样子展示出来。

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
# 输入准备好的随机样本x和其中一种类别的样本个数n
def show(n,x):
    # 定义绘图区
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    # 绘制空间中的随机样本点
    ax.scatter(x[0:n, 0], x[0:n, 1], x[0:n, 2], c='b',
        marker='x')
    ax.scatter(x[n:, 0], x[n:, 1], x[n:, 2], c='r', marker
        ='o')
    # 定义和绘制数轴图例
    ax.set_xlabel('$x^{(1)}$ Feature', color='r')
    ax.set_ylabel('$x^{(2)}$ Feature', color='r')
    ax.set_zlabel('$x^{(3)}$ Feature', color='r')
    # 显示图像
    plt.show()
```

接下来，就可以用感知机模型来处理准备好的样本数据。

1.4.3 实现感知机算法

上面生成的实例点是线性可分的，划分平面不只一个，而是属于一个集合。数据准备使用的参数 $\boldsymbol{w}\_$ 和 $b\_$ 定义了集合中的一个平面。接下来，用感知机模型找出能划分正负实例点的另一个平面。

首先，初始化平面的参数，设置学习率 η 为 0.1。

```
# 需要在空间中搜索的平面，先初始化平面的参数
W = [0, 0, 0]
b = 0
# 定义学习率参数
LRT = 0.1
```

然后遍历准备好的样例点，用感知机模型得到输出：

$$y_i = \boldsymbol{w} \cdot \boldsymbol{x}_i + b$$

```
 # 遍历样本点
for i in range(m+n):
    # 用感知机模型得到输出
    y = np.dot(w, x[i]) + b
...
```

每个实例点，已经知道了自己的正确输出分类 $y_i\_ = \{+1, -1\}$。

当 $-(y_i\_)(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) > 0$ 时，x_i 是一个误分类点，即可用学习率 η 来更新模型参数：

$$\boldsymbol{w} = \boldsymbol{w} + \eta \boldsymbol{x}_i y_i$$

$$b = b + \eta y_i$$

在遍历循环体内，使用选取到的误分类点更新平面的参数：

```
# 判断是否是误分类点
if y_[i] * y <= 0:
# 更新参数值
w += lrt * y_[i] * x[i]
```

```
5  b += lrt * y_[i]
```

更新参数值后，使用更新后的模型参数值再次遍历数据样本，查找下一个误分类点，继续更新参数值。

迭代上述步骤，直到所有实例点都被正确分类，即找出了一个可正确划分样本的平面。

记录每轮迭代的结果，可以观察到参数更新的过程，如下所示。

```
1  init w,b
2  steps:1 w:[6.778165 3.117959 -5.162033] b:0.100000
3  steps:2 w:[9.478245 10.081393 2.956793] b:0.200000
4  steps:3 w:[5.465650 6.977585 17.154110] b:0.100000
5  ...
6  steps:182 w:[115.742020 -63.439137 70.313812] b:4.200000
7  steps:183 w:[123.879997 -63.232628 55.034077] b:4.300000
8  Success!
```

由于平面可能有多个，如果把顺序遍历改为随机打乱次序后再遍历，则每次执行可能会找到一个不同的平面，这些平面都能对样本点做正确划分。

可以绘制这些平面在空间中的图像，直观地了解模型效果。

先定义一个函数：通过平面的参数，生成空间中平面上点的坐标。

```
1  # 通过平面的参数，生成空间中平面上点的坐标。
2  def getHypePlane(w, b):
3      # 用0到100之间，步长为1的数字序列，构成一个向量
4      vec = np.arange(0, 100, 1)
5      # 使用这个向量，进一步生成空间中平面前两个维度的网格矩
           阵
6      x, y = np.meshgrid(vec, vec)
7      # 用平面的参数和前两个维度，反推出第3维的网格矩阵
8      z =  -(w[0] * x1 + w[1] * x2 + b) / w[2]
9      # 返回定义超平面的三元组
10     hp=(x, y, z)
11     return hp
```

把生成的平面网格坐标作为一个新增参数，送给在绘制空间中样例点的 show() 方法。

```
# show()方法新增了hp参数，用于接收参数绘制平面
def show(n, x, hp):
    # 定义绘图区
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ...
    # 绘制平面，rstride和cstride分别定义行和列网格的步长
    # alpha定义平面透明度，cmap定义颜色渐变模式
    ax.plot_surface(hp[0], hp[1], hp[2], rstride=1,
        cstride=1, alpha=0.8, cmap=plt.cm.coolwarm)
    ...
    plt.show()
```

灵活运用基本的绘图方法，可以简便地绘制出深度学习模型中参数优化和训练指标的图像，有助于直观理解抽象的算法概念。

1.5 小结

感知机模型是神经网络的理论基石之一。这一章，我们用感知机模型解决了一个目标问题，即 D 维空间内，N 个线性可分实例点的二分类问题；还简要介绍了环境搭建的建议方法；然后描述了数据准备，模型算法的实现，最后观察了模型的训练过程。

感知机模型解决了二分类问题，现实世界中，更常见的是多类别归属的划分问题，这个方法是否能够处理呢？

答案是肯定的。下一章，我们对感知机模型加以改进、推广到多分类的场景，并使用改进后的算法构造模型、训练参数，实现手写数字的识别，得到多分类场景下识别率大于 90% 的分类模型。

参 考 文 献

[1] 李航. **统计学习方法**. 北京：清华大学出版社，2012.

[2] Rosenblatt F. **The perceptron: A probabilistic model for information storage and organization in the brain**. Psychological Review，1958，65(6)：386-408.

第 2 章

第一个神经网络

> ". . . We may have knowledge of the past and cannot control it; we may control the future but have no knowledge of it."
>
> "往者可知然不可谏，来者可追或未可知。"
>
> Claude Shannon, 1959

第 1 章提到感知机模型是一个简洁的二分类模型。在这一章里，我们把这个模型推广到**多类分类**问题，**不借助深度学习框架**，构建一个**全连接神经网络**，再应用于本章的目标问题：MNIST 手写数字识别。

2.1 目标问题：MNIST 手写数字识别

辨认识别书面形式的文字符号，以及对抽象图形、照片图像、视频进行特征识别与目标分类，是机器学习比较典型的应用场景。常见的案例包括：

- 金融机构为客户提供的线上证件号码识别与自助开户核身。
- 交通系统无所不在的牌照识别与车辆外观特征检索。
- 电子扫描文档的光学字符识别（OCR）。
- 安防领域的人脸识别与运动特征的模式匹配。

这些成功落地的应用，释放了人力资源，为企业节省了大量成本。

深度学习方法把图像信息识别的正确率提升到新的高度。这些案例背后最基本的算法原理，可以通过处理 MNIST 手写数字识别问题来展现。

2.1.1 数据集

MNIST 数据集是一套手写数字图片公开数据集，在第一版的基础数据集中，共有 7 万张手写数字图片；其中，6 万张图片用于模型训练，1 万张图片用于模型的测试与验证。验证数据集是独立于训练数据的，整套数据的格式预处理方便，适合作为验证机器学习与模式识别方法的目标问题，如图2.1所示。

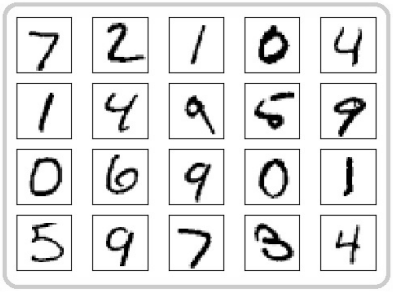

图 2.1 MINST 数据集中的手写数字

数据集中有的笔迹来自政府机构职员，相对易于辨认；也有来自数百位中学生的潦草字迹，其中一部分容易混淆的数字图片即使由人工来判断，也有一定挑战，如图2.2所示。

图 2.2　数据集中容易混淆的数字图片，
图片来自参考文献

2.1.2　图像数据和图向量

数据集中，每张图片的内容是一个手写体数字符号，表达了 0~9 这 10 个数字中的一个，数字符号处于每张图片的中间位置，大小尺寸接近，上下无颠倒，每张图片被划分为 $28 \times 28 = 784$ 个小区域，每个区域根据灰度取一个整数值，范围在 [0,255] 之间；如果把每个小区域看成图片的一个特征，每张图片看成具有 784 个特征的**图向量**, 参见图2.3，手写数字识别问题就成为：**根据 784 个维度的特征，对图像做 10 类分类的问题**。

2.2　挑战：从二分类到多分类

对类别 $K > 2$ 的多分类场景，一种思路是把它视为 $K-1$ 个二分类问题：第一次，把样本数据集的某一个类别，和余下的 $K-1$ 类（合并成一个大类）做二类划分，识别出第一个类别；依此类推，第 i 次，划分第 i 类和余下的 $K-i$ 类；经过 $K-1$ 次迭代，最终完成全部样本归属类别的划分。

这里，我们选用一种**更直接的思路**。

回忆感知机二分类模型，只包含了一个输出节点 y；现在把输出节点扩展为 K 个；权值向量 $\boldsymbol{w}$ 扩展为 $D \times K$ 的权值矩阵（weight matrix）$\boldsymbol{W}$，偏置 b 也由原来的标量扩展为长度为 K 的数组 $\boldsymbol{b}$；这样一来，表达一个 D 维样本

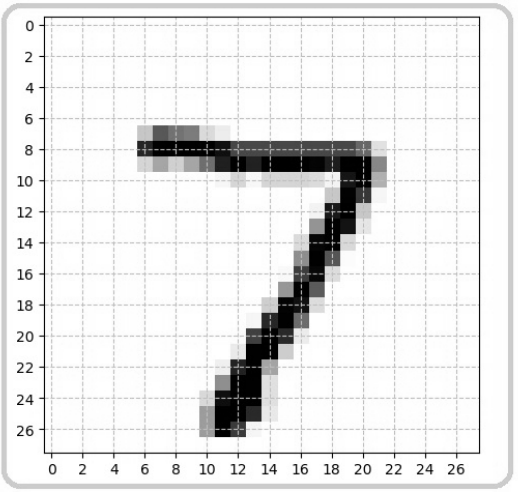

图 2.3 每个手写数字，可以划分为 784 个小区域，
继而展开成一维图向量的形式，
每个分量表达了对应小区域的灰度特征

的向量 $\boldsymbol{x}$ 经过模型处理，会得到这个样本点在所有 K 个类别上的归类可能性得分 $\boldsymbol{z}$。

$$\boldsymbol{z} = \boldsymbol{x} \cdot \boldsymbol{W}_{D\times K} + \boldsymbol{b} \tag{2.1}$$

归属可能性得分 $\boldsymbol{z}$ 是维度为 K 的向量，某一个分量 z_j 的数值越大，输入样本属于对应分类类别 j 的可能性就越高。

例如：

$$\begin{aligned}\boldsymbol{z} &= \boldsymbol{x} \cdot \boldsymbol{W}_{D\times K} + \boldsymbol{b} \\ &= [0,-4,-9] \cdot \begin{pmatrix} 3 & -9 \\ -6 & 1 \\ -4 & -7 \end{pmatrix} + [2,7] = [62,66]\end{aligned}$$

由于内积运算的性质：

$$\boldsymbol{a}^{\mathrm{T}}\boldsymbol{b} = \boldsymbol{b}^{\mathrm{T}}\boldsymbol{a}$$

等号左右两边的向量运算结果是同一个标量数值。式 (2.1) 与上式稍有不

同但结果类似，输入样本向量 $\boldsymbol{x}$ 和权值参数矩阵 $\boldsymbol{W}$ 在表达式中的先后顺序，尽管有不同的写法，输出只相差一个转置步骤，分类的数值结果是一致的。

在二分类感知机模型中，训练得到参数的几何意义是直观的，权参向量 $\boldsymbol{w}$ 是多维空间中分离超平面的法向量，偏置 b 是超平面偏离原点的截距。

在这个多分类模型中，权值参数矩阵 $\boldsymbol{W}$ 在 K 这个维度上的每一列，都可以理解为，与偏置向量 $\boldsymbol{b}$ 的对应分量一起，共同定义了一个在多维空间中的超平面，所有这些超平面共同把样本空间上的手写数字图片划分到对应类别。

另一个直观理解这个多分类模型的方式是把权值参数矩阵 $\boldsymbol{W}$ 看成是十个数字的模板原型（Prototype），用待分类图片和模板做内积运算，根据运算结果，确定和输入图片最匹配的一个模板类别，可类比为用输入图片和模板做最邻近（Nearest Neighbor）分析, 不同的是这里的模板是训练得到的。观察训练过程中 $\boldsymbol{W}$ 的变化情况如图2.4所示可以直观地理解模板捕获手写数字特征的渐变过程。

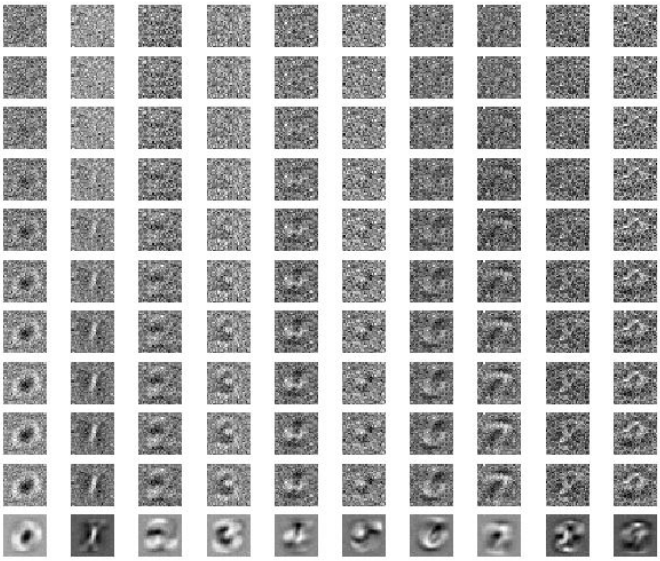

图 2.4　权值参数可以被理解分类模板，
模板内容是随机初始化后训练得到的；
训练过程中，权值参数逐渐捕捉到数字图片的特征

2.3 Softmax 方法

Softmax 方法把输出归属可能性得分 $\boldsymbol{z}$ 进一步转换，得到某个样本点归属于各个类别概率分布的形式。

例如，归属于类别 j 的概率为

$$p_j = \text{softmax}(z_j) = \frac{\mathrm{e}^{z_j}}{\sum_{k=1}^{K} \mathrm{e}^{z_k}} \tag{2.2}$$

在所有类别上的概率分布为

$$\boldsymbol{p} = \text{softmax}(\boldsymbol{z}) = \left[\frac{\mathrm{e}^{z_1}}{\sum_{k=1}^{K} \mathrm{e}^{z_k}}, \frac{\mathrm{e}^{z_2}}{\sum_{k=1}^{K} \mathrm{e}^{z_k}}, \cdots, \frac{\mathrm{e}^{z_k}}{\sum_{k=1}^{K} \mathrm{e}^{z_k}} \right] \tag{2.3}$$

这个结果在形式上可以理解为，推理结果在各个类别上的概率分布：在所有类别上的输出概率都不小 0，且所有类别上的输出概率和等于 1。

例如:

$$\begin{aligned} \boldsymbol{p} &= \text{softmax}(\boldsymbol{z}) = \text{softmax}([62, 66]) \\ &= [0.02, 0.98] \end{aligned}$$

经过这一步骤，得到了模型预测输出结果在各个类别上的概率分布值。

有的文献材料会出现“Softmax 层参数”的提法，然而观察 Softmax 函数本身，并没有参数需要训练，也没有超参数需要设置，为什么会出现这样的提法呢？

在深层神经网络结构中，不同的处理层，输入和输出数据的维度往往不同；而 Softmax 函数用来把模型的分类输出转换为概率分布形式，并不改变数据的维度，这就需要 Softmax 函数处理之前，有一个专门的处理层把输入数据的维度转换为输出分类的维度。

在这一章目标问题的例子里，这个专门的处理层就是式 (2.1) 表达的全连接层，假如送入这个全连接层的数据不是一个 mini-batch，仅仅是单个样本，如一张手写数字图片，输入数据的规格维度应该是（1,D）。

$$\boldsymbol{z} = \boldsymbol{x}_{1\times D} \cdot \boldsymbol{W}_{D\times K} + \boldsymbol{b}$$

介绍 MNIST 数据集的时候，我们知道图片被看成 $28 \times 28 = 784$ 像素的图向量，与维度 $(784, 10)$ 的权值参数矩阵 $\boldsymbol{W}$ 做内积运算之后得到 $(1, K)$ 的运算结果，这个结果与同为 $(1, K)$ 维度的偏置参数向量 $\boldsymbol{b}$ 相加，得到全连接层的输出 $\boldsymbol{z}$, 维度也是 $(1, K)$，对应这个输入样本在 K 个分类上的类别归属得分，接下来才可以送到 Softmax 函数做概率分布转换，得到归属类别上概率分布的形式。

$$P = \text{softmax}(\boldsymbol{z}) = \text{softmax}(\boldsymbol{x} \cdot \boldsymbol{W} + \boldsymbol{b}) \tag{2.4}$$

这两步处理有时被合在一起理解为 Softmax 回归模型，此时模型中最后一个全连接层的参数，被称为 Softmax 模型的参数，或者简称 Softmax 参数。

注：Softmax 模型和**多项对数几率回归模型**（Multi-norminal Logistic Regression model）有不同的表达形式，都是符合指数分布族（Exponential Family of Probability Distribution）的模型，可以纳入**广义线性模型**（Generalized Linear model）框架下完成参数求解。

为了描述不同环节算法的分解处理步骤，本书中所指的 Softmax 处理，是不包含全连接层的 Softmax 函数处理的，因而不包含需要训练的参数。

2.4 正确分类的独热编码

为了和推理结果做量化比较，数据集中每个样本的正确分类标注（Ground Truth labels）也要通过简单的编码转换为 K 维向量。

在目标问题中，每张图片所代表的正确数字可以视为一个概率分布：落在正确类别上的概率为 1，其他类别上为 0。

例如，数字 9 的图片所对应的正确标注为（0,0,0,0,0,0,0,0,0,1）。而数字 1 的图片所对应的正确标注为（0,1,0,0,0,0,0,0,0,0）。

这些正确标注可视为代表图片正确分类的向量 $\hat{\boldsymbol{y}}$ 转置为行向量的形式。这种形式也被称为**独热**（**one-hot**）编码。

在机器学习领域中，独热编码是把数据的离散型特征，比如分类的归属类别，扩展到欧氏空间的常用方法。经过独热编码处理，具有离散特征的数据被向量化为欧氏空间上的点，特征的数量就是欧氏空间上点的维度。当数据的特征量大，转化为向量后维度较高时，还可以使用**主成分分析**（Principal Components Analysis，PCA）方法做降维处理。

当前场景下，手写数字的正确标注经过独热编码转化为向量后，只有 10 个维度，不需要继续做降维处理。

有了预测输出和正确答案的概率分布，就可以通过刻画两者之间的相似度来简便地量化模型预测的损失。

2.5 损失函数——交叉熵

经过 Softmax 转换为概率分布的预测输出 p，与正确类别标注 $\hat{\boldsymbol{y}}$ 之间的误差损失，可以用两个概率分布的 **交叉熵**（**Cross Entropy**）来表达：

$$H(p_{\hat{\boldsymbol{y}}}, p) = -\sum_{j=1}^{K} p_{\hat{y}_j} \log(p_j) \tag{2.5}$$

由于正确分类采用了独热编码后的向量形式，展开式 (2.3) 后，只留下对应正确分类的非零求和项，所以某一样本点使用模型预测的损失函数可以写为

$$\text{Loss}(\hat{\boldsymbol{y}}) = -\sum_{j=1}^{K} p_{\hat{y}_j} \log(p_j) \tag{2.6}$$

$$= -\log\ \mathrm{e}^{\boldsymbol{z}_j} + \log \sum_{k=1}^{K} \mathrm{e}^{z_k} \tag{2.7}$$

你可以跳过关于交叉熵的展开介绍，从**学习算法**处继续阅读，不影响方法使用。

2.6 信息熵和交叉熵

2.6.1 信息熵

1948 年，Claude E Shannon 首次提出**信息熵（entropy）**的概念。

假设有限个离散型随机变量 X 的概率分布是 $P(X)$，则随机变量 X 的熵定义为

$$H(P) = -\sum_{x} P(x) \log P(x) \tag{2.8}$$

在信息论和概率论范畴，熵用来表达随机变量的不确定性，熵越大，随机变量的不确定性越大。比如从一个装满小球的盒子里随机取出一个，如果盒子

里全是白色小球，即取出白色小球的概率是 1。

$$H(P) = -1 \times \log_2 1 = 0$$

此时熵为 0，取出小球的颜色是确定的。

如果盒子里白色和红色小球各占一半，随机取出一个，采取“有放回抽样法”重复取出多次，取出白色小球次数占总次数的比例会向一半靠近，概率为 0.5 时：

$$H(P) = -0.5 \times \log_2 \frac{1}{2} - 0.5 \times \log_2 \frac{1}{2} = 1$$

熵达到最大值 1，此时，每次取出小球颜色的确定性最低。

如果盒子里放的红色小球比例继续提高，熵也随之下降，当盒子里全是红色小球的时候，熵重新降回到 0，每次取出的颜色又是确定的。

信息熵是机器学习中的常用度量方法，在决策树（decision tree）的特征选择，最大熵模型（Maximum Entropy Model，MEM）的基本原理解释中都有应用。

2.6.2 交叉熵

交叉熵（Cross Entropy）的概念同样可以从信息论的视角来理解：若离散事件以真实概率 $p(x_i)$ 分布，则以隐概率分布 $q(x_i)$ 对一系列随机事件 x_i 做最短编码，所需要的平均比特 (bits) 长度，即可用交叉熵来表达。

其中，定义 $q(x_i) = 1/2^{l_i}$，显然，较短的编码长度 l_i 应当被用于出现概率 $q(x_i)$ 较高的编码片段，以增加信息密度、提高传输效率。

$$\begin{aligned} H(p,q) =& E_p[l_i] && (2.9)\\ =& E_p[\log_2(\frac{1}{q(x_i)})] && (2.10)\\ =& \sum_{x_i} p(x_i) \log_2 \frac{1}{q(x_i)} && (2.11)\\ =& -\sum_i p_i \log_2(q_i) && (2.12) \end{aligned}$$

直观理解，如果有 $H(p,q') < H(p,q)$, 则相对于 q_i , 概率分布 q_i' 同真实

概率分布 p_i 更相似。

交叉熵对两个概率分布的差异描述不具**对称性（Symmetry）**，所以交叉熵并不是严格意义上的**距离**。

如果看到“交叉熵描述两个概率分布的距离”，或者“交叉熵等价于最小化 KL 距离”这样的表述，应该理解为本义是指“交叉熵刻画两个概率分布的差异”，因为 KL 散度同样不具对称性，也不是严格意义的距离。

从交叉熵概念的源头看，是用比特（bit）信息为单位，因而以 2 为底做对数计算，那么用损失函数计算 Loss 时，对数计算**是否必须以 2 为底**呢？

答案是否定的。

在机器学习领域中，交叉熵被用来衡量**两个概率分布的相似程度**，交叉熵越小，两个概率分布越相似。实践中，出于**简化推导、优化数值计算效率**的考虑，对数的底可以因地制宜地进行其他选择。

例如以 e 为底的情况，有换底公式：

$$\ln(q_i) = \frac{\log_2 q_i}{\log_2 \mathrm{e}}$$

可知，对数的底由 2 换成 e，对损失 Loss 的影响是，缩小了常数倍 $\log_2 \mathrm{e}$；优化损失函数使损失极小的场景下，函数在同一权重参数处取得极值，因而函数取值的数值**缩放正倍数**不影响优化方法。

所以损失函数式 (2.7) 也可以写为

$$\mathrm{Loss}(\hat{\boldsymbol{y}}) = -\boldsymbol{z}_j + \ln \sum_{k=1}^{K} \mathrm{e}^{z_k} \tag{2.13}$$

这个结果把式 (2.7) 右边第一项 $-\log\ \mathrm{e}^{z_j}$ 中的对数运算及 e 指数运算两个步骤转换为一步求和计算，使学习算法中参数求解的推导步骤更简洁。实践中，由于简化了误差损失和参数梯度的运算逻辑，还可以提高整个模型的训练效率。

2.7　第一个神经网络的学习算法

先为模型参数 $\boldsymbol{W}, \boldsymbol{b}$ 设置初始值，$\boldsymbol{W}$ 随机初始化为较小的值，$\boldsymbol{b}$ 初始化为 0；然后用上一章讨论的梯度下降法（Gradient Descent），不断用训练得到的交叉熵损失计算回传梯度 $\nabla \boldsymbol{W}, \nabla \boldsymbol{b}$，继而更新参数来极小化损失函数，迭

代得到参数训练结果，如图2.5所示。

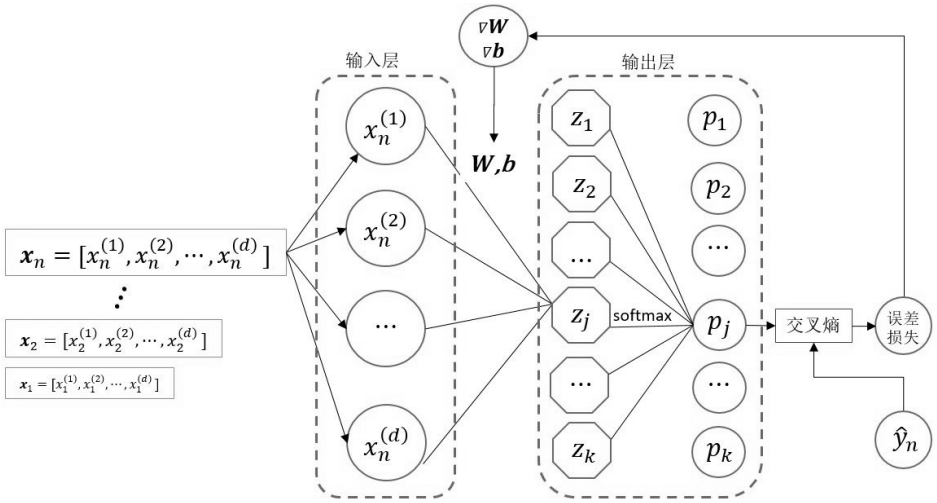

图 2.5　全连接神经网络的结构：通过交叉熵获得训练损失，反向传播回传给各处理层，计算各层参数的梯度，更新得到新的参数值

对包含 D 维输入特征的 K 类分类样本点，根据损失函数计算参数更新的梯度：

$$\nabla_{w_{dj}}\text{Loss} = \frac{\partial \text{Loss}}{\partial w_{dj}} \tag{2.14}$$

$$= \frac{\partial(-\boldsymbol{z}_j + \ln \sum_{k=1}^{K} \mathrm{e}^{z_k})}{\partial w_{dj}} \tag{2.15}$$

对式 (2.15) 的前一部分 $-\dfrac{\partial(\boldsymbol{z}_j)}{\partial w_{dj}}$，将 $\boldsymbol{z}_j$ 中向量和矩阵的乘积运算展开，得到：

$$-\frac{\partial(\boldsymbol{x}\cdot\boldsymbol{W}+\boldsymbol{b})}{\partial w_{dj}} = \begin{cases} -x^{(d)} & , \ j = k \\ 0 & , \ j \neq k \end{cases} \text{即} -p_{\hat{y}_j} x^{(d)} \tag{2.16}$$

对后一部分，应用链式法则：

$$\frac{\partial(\ln\sum_{k=1}^{K}\mathrm{e}^{z_k})}{\partial w_{dj}}=\frac{1}{\sum_{j=1}^{K}\mathrm{e}^{z_j}}\cdot\frac{\partial\sum_{j=1}^{K}\mathrm{e}^{z_j}}{\partial w_{dj}} \tag{2.17}$$

$$=\frac{\mathrm{e}^{z_j}}{\sum_{j=1}^{K}\mathrm{e}^{z_j}}\cdot\frac{\partial\mathrm{e}^{z_j}}{\partial w_{dj}} \tag{2.18}$$

$$=p_j x^{(d)} \tag{2.19}$$

两部分合并得到：

$$\nabla_{w_{dj}}\mathrm{Loss}=\frac{\partial\mathrm{Loss}}{\partial w_{dj}}=(p_j-p_{\hat{y}_j})x^{(d)} \tag{2.20}$$

同样：

$$\nabla_{b_j}\mathrm{Loss}=\frac{\partial\mathrm{Loss}}{\partial b_j}=p_j-p_{\hat{y}_j} \tag{2.21}$$

从而得到参数更新的梯度。其中 $p_{\hat{y}}$ 的分量 $p_{\hat{y}_j}$ 的值，由 j 是否等于正确类别 k 而定。

$$\nabla\boldsymbol{W}=\boldsymbol{x}^{\mathrm{T}}\cdot(p-p_{\hat{y}}) \tag{2.22}$$

$$\nabla\boldsymbol{b}=\sum_{j=1}^{K}(p-p_{\hat{y}}) \tag{2.23}$$

$$p_{\hat{y}_j}=\begin{cases}1 & ,\ j=k\\ 0 & ,\ j\neq k\end{cases} \tag{2.24}$$

再用梯度更新模型参数：

$$\boldsymbol{W}=\boldsymbol{W}+\eta\nabla\boldsymbol{W} \tag{2.25}$$

$$\boldsymbol{b}=\boldsymbol{b}+\eta\nabla\boldsymbol{b} \tag{2.26}$$

以上推导完成了第一个神经网络模型中全部训练参数的梯度计算与更新。

在深度学习领域中，上述处理是反向传播的关键步骤。

2.8 反向传播

反向传播（Back Propagation）是相对**前向传播（Forward Propagation）**过程而言的；从数据集中抽取训练数据，首先使用算法前向传播得到分类结果，再结合训练数据的正确标注，计算样本预测损失，然后根据损失更新神经网络模型参数，每次迭代训练后释放训练得到的分类结果，只保留更新过的参数。这个迭代过程就是**反向传播**。

把独立样本送入模型，使用训练得到的模型参数，做验证或者预测的步骤，常被称为**推理预测（Inference）**；全部训练完成后，使用独立数据集对参数结果做推理预测，用于验证和评价模型的训练结果；在训练期间往往也需要使用独立数据集对模型做跟踪验证，及时观察模型的动态收敛情况，以便发现过拟合等问题时能及时停止训练（Early Stopping）。

实践中，往往从训练样本集里随机抽取一批训练样本（mini-batch），通过对整批数据进行矩阵运算，得到这批样本损失的均值，以此减少更新梯度的迭代次数，提高训练效率。每次迭代训练后，使用该批次的梯度均值更新一次参数，相对于逐样本迭代，整批训练迭代能较快得到接近单样本梯度下降的收敛结果。实践中，在内存或显存容量允许的情况下，可以尝试用较大的 mini-batch 得到更好的训练效果。

$$\text{Param} = \text{Param} + \eta \frac{\nabla \boldsymbol{W}}{\text{batch capacity}}$$

采用无放回抽样，一次随机抽取一个 mini-batch 作为训练数据，直到全部训练样本都被抽样到，即完成一轮（Epoch）训练。复杂的深度学习模型，可能需要多轮来训练模型参数。

预测推理的时候，只需要向模型送入待预测的样本，待预测的样本可能只是单张图片，送入模型做前向计算，得出推理分类结果。对于解决本章目标问题的第一个神经网络模型，这个过程只需要做一次向量与矩阵的乘法运算和一次向量相加这两步简单运算，即可快速得到预测推理的分类结果。

主流的深度学习框架，通过自动微分来支持反向传播，为什么我们还要了解反向传播的原理呢？

首要原因是，反向传播存在**抽象泄漏（Leaky Abstractions）**问题。

2.9 抽象泄漏

> “All non-trivial abstractions, to some degree, are leaky.”
> “所有的复杂抽象，或多或少，都有泄漏。”
>
> Joel Spolsky, Co-funder of Stack Overflow, 2002

抽象泄漏法则告诉我们：所有试图隐藏复杂细节的抽象，都不能做到完全封装，总会有底层机制泄漏到抽象之上，给使用者带来麻烦；表层抽象，初期节约了使用成本，然而长远看，这些漏洞会以各种“灵异缺陷”或者“性能问题”的形式呈现出来。要解决这些漏洞，则需要深入了解抽象之下的原理。

Joel Spolsky 用“好莱坞速运”的例子，形象地类比了网络传输协议栈中，TCP 对 IP 的抽象泄漏：

> “假设我们要把演员从纽约百老汇送到西海岸的好莱坞，采取的方法是用汽车载着演员横穿整个大陆。演员们可能在路上遭遇车祸，不幸去世；也可能在途中喝醉酒，跑去剃了光头，或是纹了刺青，总之丑到无法工作的地步；此外，演员们走的路线常常不同，可能后出发的演员反而先到达。
>
> 现在我们有了一项新服务：“好莱坞速运”，能够确保演员们以（a）良好状态（b）依次（c）到达。诀窍是不再去关注、应对旅途上各种不可靠状况，而是在终点确认每位演员是否状态良好地按时到达，否则就呼叫总部，要求送来演员的双胞胎。如果演员虽然能到达，但是顺序不对，就先排好队再露面。这样，就算有架 51 区大飞碟坠毁在内华达州的主干道上，阻塞了交通，迫使所有在途演员都要改道亚利桑那州，加利福尼亚州的导演也永远听不到什么飞碟坠毁事件，因为在导演看来，演员们只是比平常晚到了一会儿。”

同样的，TCP 号称可靠，然而并非如此，因为所依赖的 IP 层及以下协议，原本就不是 100% 可靠的。首先，物理线路不通，任何 IP 包都不能通过，TCP 也不会工作；更常见的情况是，交换机超负荷工作，IP 包仅能部分通过，TCP 看上去还在工作，却慢得不得了。TCP 试图对不可靠的底层提供完整的抽象，却不能真正保护使用者免受底层问题的影响。

本书会讨论深度学习算法中典型的抽象泄漏场景。了解这些算法，包括反向传播算法的原理，可以更好地理解相关的抽象泄漏问题，直达本质，更有效

地使用构建在算法之上的深度学习框架。

2.10 算法实现

2.10.1 数据准备

可以从 Yann LeCun 的网站`http://yann.lecun.com/exdb/mnist`下载这套数据集，共四个文件包：

- train-images-idx3-ubyte.gz: 训练图片集（9912422 bytes）。
- train-labels-idx1-ubyte.gz: 训练图片集的正确标注（28881 bytes）。
- t10k-images-idx3-ubyte.gz: 测试图片（1648877 bytes）。
- t10k-labels-idx1-ubyte.gz: 测试图片的正确标注（4542 bytes）。

下载文件包并解压缩后，得到二进制格式的图片和标注文件，图片文件中所包含的每一张手写数字图片都对应标注文件中一个正确归类的数字标注。

文件内部的图片不是看图工具能直接打开的常见格式，因此送入模型处理前需要先把图片信息按照文件规格读取出来。

MNIST 文件的格式

四个文件包中的文件格式一共分为图片文件 *-images-* 和标注文件 *-labels-* 两种；图片文件的前 16 个字节是文件格式、图片数量、规格的描述，图片的像素信息从第 17 个字节开始。

[offset] [type] [value] [description]

0000 32 bit integer 0x00000803(2051) magic number
0004 32 bit integer 60000 number of images
0008 32 bit integer 28 number of rows
0012 32 bit integer 28 number of columns
0016 unsigned byte 0 pixel
0017 unsigned byte 0 pixel
.....
xxxx unsigned byte xx pixel

标注文件 labels 略有不同，前 8 个字节是对文件格式和标注数量的描述；而表述正确分类的标注信息则从第 9 个字节开始。

[offset] [type] [value] [description]

0000 32 bit integer 0x00000801(2049) magic number

0004 32 bit integer 60000 number of items

0008 unsigned byte 7 label

0009 unsigned byte 2 label

.....

xxxx unsigned byte xx label

理解了文件格式，就可以很容易地读取 MNIST 数据。

读取 MNIST 数据

定义下载和解压缩 MNIST 数据的目录 `your_unpacked_MNIST_DATA_path`。

```
1 import struct
2 import os
3 import numpy as np
4 # 定义数据文件路径
5 path=your_unpacked_MNIST_DATA_path
6 kind=train
7 images_path = os.path.join(path,'%s-images.idx3-ubyte' %
      kind)
8 labels_path = os.path.join(path,'%s-labels.idx1-ubyte' %
      kind)
```

从第 9 个字节开始加载正确标注信息，获取的全部正确标注 labels 的总数量等于图片的总张数。

```
1 with open(labels_path,'rb') as labelfile:
2     magic, n = struct.unpack('>II',labelfile.read(8))
3     labels = np.fromfile(labelfile,dtype=np.uint8)
```

加载从第 17 个字节开始的图片像素数据，再把每张 28×28 像素的图片都拉伸（reshape）为 784×1 的图向量。

```
with open(images_path,'rb') as imagefile:
    magic, num, rows, cols = struct.unpack('>IIII',
        imagefile.read(16))
    images = np.fromfile(imagefile,dtype=np.uint8).reshape
        (len(labels),784)
```

对图像数据 image 的每一个样本图片做简单的尺度调整处理，方便数值运算，这个尺度调整不是缩小图片的尺寸，而是把图向量每个特征分量上表达灰度的数值做比例缩小：

```
images = images/255
```

为了确保模型训练效果，需要随机打乱读取的数据次序，再按照 mini-batch 的容量，不回放地抽取全部训练数据，送给模型进行训练；每完成一轮训练后，再按照随机顺序，重新组织全部 mini-batch 分组，在下一轮用每个 mini-batch 迭代训练。

MINIST 数据的载入逻辑，以及每轮训练前对全部训练数据所的随机重新分组的处理，都可以放在一个数据预处理类内部完成。

```
import numpy as np
import random
import struct
import sys
import os
# MNIST数据预处理类
class MnistData(object):
    def __init__(self, absPath, is4Cnn, dataType):
        self.absPath = absPath
        # True for cnn,False for other nn structures
        # CNN的数据规格不同，此处将is4Cnn参数设置为False
        self.is4Cnn = is4Cnn
        # 数据类型统一定义，便于灵活调整，此处定义为np.
            float32
        self.dataType = dataType
        self.imgs, self.labels = self._load_mnist_data(
            kind='train')
```

```
16          self.imgs_v, self.labels_v = self._load_mnist_data
                (kind='t10k')
17          # 训练样本范围
18          self.sample_range = [i for i in range(len(self.
                labels))]
19          # 验证样本范围
20          self.sample_range_v = [i for i in range(len(self.
                labels_v))]
21
22      # 加载MNIST数据集
23      def _load_mnist_data(self, kind='train'):
24          labels_path = os.path.join(self.absPath, '%s-
                labels.idx1-ubyte' % kind)
25          images_path = os.path.join(self.absPath, '%s-
                images.idx3-ubyte' % kind)
26          # 读取标注信息
27          with open(labels_path, 'rb') as labelfile:
28              # 读取前8个字节
29              magic, n = struct.unpack('>II', labelfile.read
                    (8))
30              # 余下的数据读到标注数组中
31              labels = np.fromfile(labelfile, dtype=np.uint8
                    )
32          # 读取图片信息
33          with open(images_path, 'rb') as imagefile:
34              # 读取前16个字节
35              magic, num, rows, cols = struct.unpack('>IIII'
                    , imagefile.read(16))
36              # 余下数据读到image二维数组中, 28×28=784像素的
                    图片
37              # 共60000张(和标注项数一致)
38              # 从原数组创建一个改变尺寸的新数组(28×28图片拉
                    伸
39              # 为784×1的数组)
40              # CNN处理的输入则拉伸为28×28×1
41              if False == self.is4Cnn:
42                  images_ori = np.fromfile(imagefile, dtype=
```

```
                    np.uint8).reshape(len(labels), 784)
            else:
            # 支持多通道, 此处通道为1
                images_ori = np.fromfile(imagefile, dtype=
                    np.uint8).reshape(len(labels), 1, 28,
                    28)
        # 数值尺度调整化
        images = images_ori / 255
        return images, labels

    # 对训练样本序号随机分组
    def getTrainRanges(self, miniBatchSize):
        # 定义样本范围
        rangeAll = self.sample_range
        # 随机打乱顺序号
        random.shuffle(rangeAll)
        # 按照mini_batch大小, 对随机乱序的顺序号分组
        rngs = [rangeAll[i:i + miniBatchSize] for i in
            range(0, len(rangeAll), miniBatchSize)]
        return rngs

    # 获取训练样本范围对应的图片和标注
    def getTrainDataByRng(self, rng):
        # 按照传入的一个分组, 获取一个mini_batch的图向量
        xs = np.array([self.imgs[sample] for sample in rng
            ], self.dataType)
        # 获取这组图向量对应的正确类别标注
        values = np.array([self.labels[sample] for sample
            in rng])
        return xs, values

    # 获取随机验证样本
    def getValData(self, valCapacity):
        samples_v = random.sample(self.sample_range_v,
            valCapacity)
        #  验证输入 N×28×28
        images_v = np.array([self.imgs_v[sample_v] for
```

```
            sample_v in samples_v], dtype=self.dataType)
        #  正确类别 1×K
        labels_v = np.array([self.labels_v[sample_v] for
            sample_v in samples_v])

    return images_v, labels_v
```

经过上述数据预处理，我们可以得到训练数据集的图向量数据 **images** 和正确答案数组 **labels**，以及验证用图片 **images_v** 和正确答案 **labels_v**。

每轮训练之前，将先打乱顺序再重新划分的 mini-batch 送入模型进行下一轮训练。

2.10.2 实现第一个神经网络

这是一个单层神经网络模型，在 MNIST 手写数字识别问题场景下，需要训练的参数是权值矩阵 $\boldsymbol{W}$（784×10）和偏置向量 $\boldsymbol{b}$（1×10），首先做两组参数的初始化，并预设 mini-batch 容量为 256，学习率设置为 0.1：

```
batch_size = 256
LEARNING_RATE = 0.1
# D×K
w = 0.01 * np.random.randn(784,10)
# 1×K
b = np.zeros(10)
```

随机抽取一批（mini-batch）训练图片 $\boldsymbol{x}$：256×784 作为输入送入模型，得到类别归属可能性得分 $\boldsymbol{z}$：

$$\boldsymbol{z} = \boldsymbol{x} \cdot \boldsymbol{W} + \boldsymbol{b}$$

使用 Softmax 函数把输出进一步转换为概率分布：

$$\boldsymbol{p} = \text{softmax}(\boldsymbol{z}) = \frac{\mathrm{e}^{z_j}}{\sum_{k=1}^{K} \mathrm{e}^{z_k}} \quad (j = 1, 2, \cdots, K)$$

实践中，指数计算的结果可能会是很大的数值，需要通过将 Softmax 原始输入做预处理来防止指数计算的结果做大数相除时越界溢出。观察 Softmax 函

数，给分数的上下两部分同乘一个常系数 C，结果是不变的，可以借助指数运算的特性，把包含常系数的乘法步骤变成 e 指数的加数项 $\ln C$，这个加数项仍然是个常数。

$$\frac{\mathrm{e}^{z_j}}{\sum_{k=1}^{K}\mathrm{e}^{z_k}}=\frac{C\mathrm{e}^{z_j}}{C\sum_{k=1}^{K}\mathrm{e}^{z_k}}=\frac{\mathrm{e}^{z_j+\ln C}}{\sum_{k=1}^{K}\mathrm{e}^{z_j+\ln C}}$$

这个常系数 $\ln C$ 取任意值都不改变 Softmax 的计算结果，这里取 $\boldsymbol{z}$ 这个归类得分向量里最大得分值的相反数：

$$\ln C=-\max\left(\boldsymbol{z}_j\right)$$

这一步运算的结果，是让归类得分向量 $\boldsymbol{z}$ 中每个分类上的分值都减去了最大分值，原来的最大分值则变成了 0，避免出现很大的指数运算结果，同时又保持 Softmax 函数输出在各个归属类别上的概率分布结果不变。

```
def softmax(z):
    #对每一行: 所有元素减去该行的最大的元素,
    #避免exp溢出,得到1×N矩阵
    max_z = np.max(z,axis=1)
    # 极大值重构为N × 1 数组
    max_z.shape=(-1,1)
    # 每列都减去该列最大值
    z1 = z - max_z
    # 计算exp
    exp_z = np.exp(z1)
    # 按行求和, 得1×N累加和数组
    sigma_z = np.sum(exp_z,axis = 1)
    # 累加和拉伸为N×1 数组
    sigma_z.shape=(-1,1)
    # 计算Softmax得到N×K矩阵
    y = exp_y/sigma_z

    return y
```

接下来，把经过 Softmax 函数处理的前向传播输出，以及对应图片的正

确独热标注，一起送入交叉熵函数，得到本轮训练的交叉熵损失：

$$H(p_{\hat{y}}, p) = -\sum_{j=1}^{K} p_{\hat{y}_j} \log(p_j)$$

交叉熵计算前，需要对输入做上下界裁剪，避免对数运算越界溢出及除 0 错：

```
# 计算交叉熵损失
def crossEntropy(y,y_):
    # 获取mini-batch 大小
    n = len(y)
    # 计算一个mini-batch 输入的平均损失
    return np.sum(-np.log(np.clip(y[range(n), y_],1e-10,
        None,None)))/n
```

有了交叉熵损失，就可以反向计算梯度，并迭代更新模型的参数：

$$\nabla \boldsymbol{W} = \boldsymbol{x}^{\mathrm{T}} \cdot (p - p_{\hat{y}})$$

$$\nabla \boldsymbol{b} = \sum_{j=1}^{K} (p - p_{\hat{y}})$$

其中 $p_{\hat{y}}$ 的分量 $p_{\hat{y}_j}$ 的值，由 j 是否等于正确类别 k 而定：

$$p_{\hat{y}_j} = \begin{cases} 1 & , j = k \\ 0 & , j \neq k \end{cases}$$

```
lossCE[range(batch_size), y_] -= 1
delta_y_mean = lossCE / batch_size
#参数梯度
delta_w = np.dot(x.T, delta_y_mean)
delta_b = np.sum(delta_y_mean, axis=0)
# 用参数的梯度和学习率更新w, b
w = w - LEARNING_RATE * delta_w
b = b - LEARNING_RATE * delta_b
```

训练过程中及时观察损失函数值 Loss 的变化，可以及时判断模型是否正

常收敛。

也可以跟踪观察模型的推理正确率：

$$\text{accuracy} = \frac{\text{NUM\_CORRECT}}{\text{BATCH\_SIZE}} \times 100\%$$

```
accuracy = np.mean(np.argmax(y, axis=1) == y_)
```

上面的实现，不借助深度学习框架，可以方便地使用单步调试，深入观察各个运算步骤之间数据的运算、流转过程，从算法层面直观了解单层全连接神经网络的基本结构。

2.10.3 实现 MINIST 手写数字识别

在 MINIST 数据集上，经过一定步骤的训练，观察模型在验证数据上的收敛情况和分类正确率：

```
start..
w,b inited..
epoch  0, learning_rate= 0.10000000
epoch  0,  loss_v=0.717528,  acc_v=0.853000
epoch  1,  loss_v=0.311690,  acc_v=0.913000
epoch  2,  loss_v=0.321564,  acc_v=0.905500
epoch  3,  loss_v=0.275345,  acc_v=0.918000
epoch  4,  loss_v=0.271198,  acc_v=0.921000
epoch  5,  loss_v=0.287199,  acc_v=0.917000
epoch  6,  loss_v=0.255700,  acc_v=0.929500
epoch  7,  loss_v=0.298908,  acc_v=0.922000
epoch  8,  loss_v=0.267614,  acc_v=0.924500
epoch  9,  loss_v=0.267489,  acc_v=0.922500
...
```

经过 5000 个批次迭代后，验证数据集上预测损失趋于稳定, 如图2.6所示。

这个仅有一层的神经网络在验证数据集上的分类正确率稳定在 92% 附近，如图2.7所示。

首次训练设置的超参数里，mini-batch 为 256，学习率为 0.1；实践中，

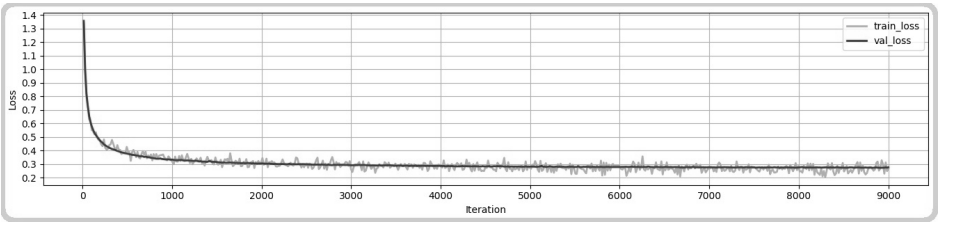

图 2.6 损失下降和模型收敛情况

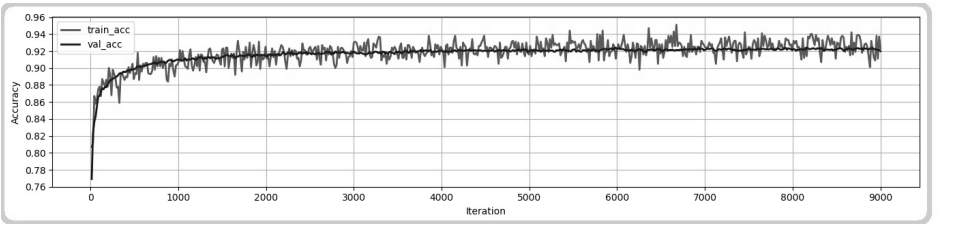

图 2.7 训练数据和独立测试数据上的分类正确率

mini-batch 容量也可以根据实际内存容量设置一个更大的值来加快模型收敛；当前场景下，数据集规模不大，模型结构也比较简单，启动训练后很快可以看到收敛结果，可以尝试把学习率以整 10 倍放大或缩小，观察不同学习率下的训练效果。

2.11 小结

这一章内容介绍了全连接神经网络的基本结构、Softmax 函数和交叉熵损失的原理，讨论了反向传播机制和抽象泄漏的概念，对训练模型的一般过程做了描述，最后不借助深度学习框架，实现了所讨论的全部算法。

这一章的最后构建了本书第一个可运行的神经网络模型。这个模型虽然结构简洁，只有一个全连接处理层，却可以在数据集上收到不错的效果。

观察训练过程，不难发现两个问题：首先，模型在训练和验证集上的表现是不一致的，训练数据集上的分类正确率高于验证数据集的；另外，训练的初期，模型表现稳步提高，经过一定轮次迭代后，指标在稳定值附近不再提升。

下一章，将介绍第一个问题：**过拟合**（Over-fitting）问题及其缓解方法；针对第二个问题，将在目前的基本结构上引入隐藏层（Hidden Layer）和激活函数，从而将模型在这个场景的预测正确率进一步提高到 98% 以上。

拓展阅读

斯坦福大学的机器学习与统计模式识别的课程（CS229），在监督学习单元，介绍了一套理论框架，把对数几率回归模型和 Softmax 方法，纳入同一套框架之下求解。

The Stanford CS class CS229

`http://cs229.stanford.edu`

Joel Spolsky 是一位软件工程师，也是 Stack Overflow 的联合创始人和 CEO，他详细讲解了抽象泄漏的危害，以及如何在实践中避免抽象泄漏。

The Law of Leaky Abstractions.

`https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions`

参 考 文 献

[1] Y LeCun, L Bottou, Y Bengio, P Haffner. **Gradient-based learning applied to document recognition.** Proceedings of the IEEE, 1998, 86(11):2278-2324.

[2] De Boer, PT, Kroese, DP, Mannor. et al. **A Tutorial on the Cross-Entropy Method.** Ann Oper Res, 2005, 134:19.

第 3 章

多层全连接神经网络

“横看成岭侧成峰，远近高低各不同。”

——[宋] 苏轼《题西林壁》

上一章，基于感知机模型构建了神经网络的基本结构，在 MNIST 的验证数据集上，得到大于 92% 的分类正确率。这一章，首先为基本结构引入**隐藏层**和**激活函数**，使模型具备更强大的特征捕获能力；再介绍**过拟合**问题和缓解方法，使这个**不借助深度学习框架**的神经网络在 MNIST 数据集上的分类正确率，进一步提高到 98% 以上。

3.1 第一个挑战：异或问题

对第 1 章中列举的三维空间中线性可分的样本点，如果把图像旋转特定角度，来观察空间中的平面，在旋转后的视角下，平面投影为一条直线，空间中的样本几乎都在这条直线的两侧。参见图 3.1所示的三维空间中线性可分的样本点，对比空间旋转前后的不同。

对这一类问题，空间中两类样本点，投影到低维度上之后，可以用一条直线进行划分，因此使用一次仿射变换，找到合适的变换参数所定义的分离超平面，就可以完成分类。

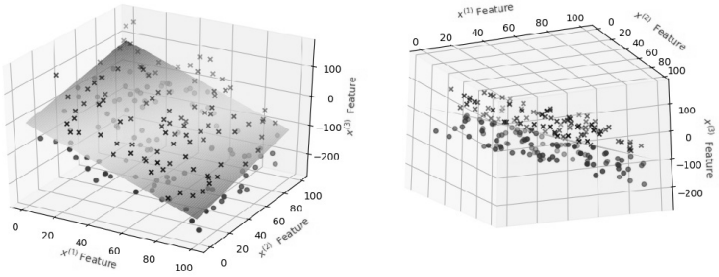

图 3.1 三维空间中线性可分的样本点，空间旋转后，
划分平面投影为一条直线，样本点的投影都在直线两侧

然而如果旋转后的视角是图3.2所示的情况，即空间中的两类样本存在**异或（XO）关系**，不能通过一个分离超平面完成划分，只做一次仿射变换的单层模型就不再适合了。为了处理异或问题，需要引入**隐藏层**（hidden layer），将模型改进成为更深的神经网络，通过多次变换，找到更多的分离平面来划分空间中的样本。

3.2 更深的神经网络——隐藏层

引入隐藏层后，原始输入要先经过隐藏层处理，再传递到输出层；隐藏层中的节点表达了从输入特征中抽取得到的更高层特征。

模型结构中增加了隐藏层之后，隐藏层的节点和输入层节点之间有了新的权值矩阵 $\boldsymbol{W}_1$ 和偏置 $\boldsymbol{b}_1$；

$$\boldsymbol{h} = \boldsymbol{x} \cdot \boldsymbol{W}_1 + \boldsymbol{b}_1 \tag{3.1}$$

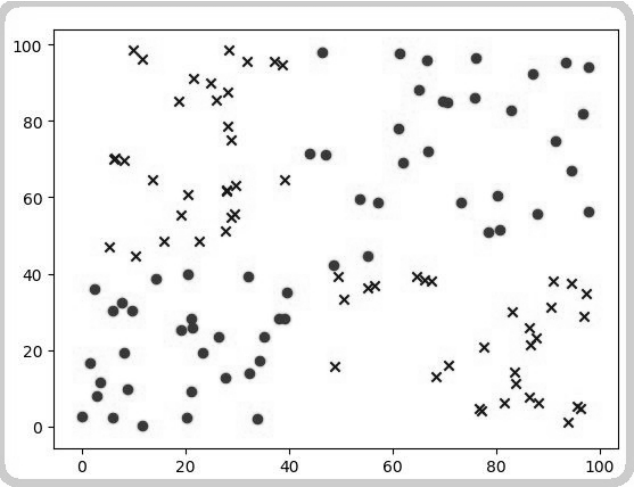

图 3.2 异或问题：空间中样本点，无法用一个平面做二类划分，需要加入隐藏层，引入更多的平面

隐藏层的输出 $\boldsymbol{h}$ 作为下一层的输入：

$$\boldsymbol{z} = \boldsymbol{h} \cdot \boldsymbol{W}_2 + \boldsymbol{b}_2 \tag{3.2}$$

增加了隐藏层之后，单层网络结构扩展成为多层结构，隐藏层和相邻处理层之间的各个节点（nodes），两两形成相互连接（edges），这样的神经网络处理层称为全连接（fully-connected，FC）层, 每一条连接都对应一个权重参数值，模型有了更多的训练参数，处理层上的每个节点都是前层全部节点的加权结果。如图3.3所示，输入数据特征维度（shape）为 8 ，中间的隐藏层有 6 个隐藏节点，输出维度为 6，整个神经网络的节点数是 19 个，节点间连接达到 78 个。

模型结构变得复杂，训练消耗的算力与时间成本也相应增加了。对于来自不同数据集，但仍属于同一种类别场景下的样本，样本的特征可能会存在一部分共性，而大规模深度神经网络模型的训练成本比较高，有时候可以尝试用一个数据集在模型中先做训练，得到一整套参数，再对结构做少许调整，然后仅仅初始化少部分参数，比如替换最后一个全连接层，并初始化最后一个全连接层的参数，但是保留并固定之前全部处理层上训练好参数。这样一来，在目标数据集上仅需要再训练最后一层重新初始化过的参数，以此实现**迁移学习（Transfer Learning）**。

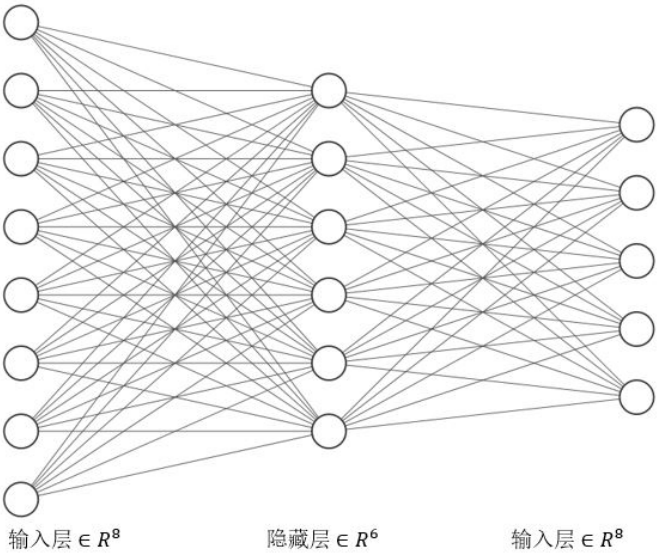

图 3.3　神经网络增加隐藏全连接层，本例中，隐藏层与相邻层共有 19 个节点，相邻层之间连接达到 78 个

更常见的情况是不去固定之前训练好的参数值，而是直接使用这些参数训练的结果作为模型的初始值，再应用于目标数据集上做增量训练。采用这种方式，有时能以较少轮次的低训练消耗，或者较小的训练数据集，得到相对理想的训练效果。

有了隐藏层之后, 单层网络结构成了多层结构，模型有了更多的训练参数，以 MNIST 数据处理模型为例，在单层网络结构下，输入数据是 784 维向量，模型的训练参数量为 $784 \times 10 + 10 = 7850$ 个。

增加了一层含有 512 个隐藏节点的隐藏层后，模型的训练参数量增加到 $784 \times 512 + 512 + 512 \times 10 + 10 = 407050$ 个。

新增加的大量参数，让模型具备了更加强大的拟合能力，当然也带来了新的挑战。

3.3　第二个挑战：参数拟合的两面性

具有大量参数的模型，会有多强大的拟合能力呢？

”With four parameters I can fit an elephant, and with five I can make him wiggle his trunk.”

“用四个参数，我能拟合出一头象，用五个参数，可以让这头象的鼻子也动起来。”

Enrico Fermi quoted Johnny von Neumann, 1953

弗里曼 · 戴森回忆，**恩瑞克 · 费米**曾经引用了**冯 · 诺伊曼**的论断：“四个参数可以拟合出大象”，来提醒他反思研究工作的物理意义和数学自洽性。

2010 年，一篇发布在《物理学期刊》（*American Journal of Physics*）上的文章，提供了用四个复参数勾勒大象的方法。参见图 3.4，用四个复参数拟合的大象。

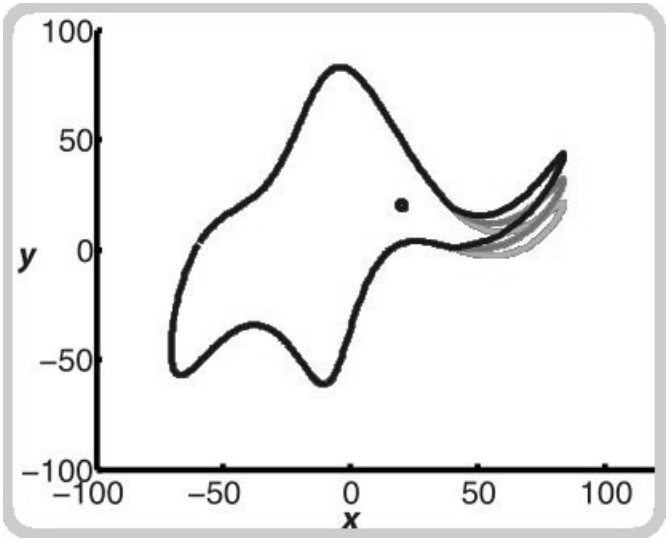

图 3.4 《物理学期刊》一篇发表于 2010 年的文章，用四个复参数拟合的大象，图片来自参考文献

这个例子生动地体现了复杂模型强大的参数拟合能力。然而在机器学习领域，使用大量参数的模型，能捕捉到训练数据集上的细微特征，并不总是有利于模型参数训练，因为训练数据集上那些不具一般性的特征，也容易被模型捕获到，继而被误当作一般特征用于测试验证，从而引发过拟合问题。

3.4 过拟合与正则化

3.4.1 欠拟合与过拟合

在训练数据集上，训练迭代次数不足，模型没有学到训练样本的一般特征，称为**欠拟合**（Under-fitting）。

反之，有大量参数的复杂模型，经过多轮训练，会将训练数据中不具一般性的噪声，也学习到模型参数里，结果在训练数据上得到很好的拟合表现，在未知数据上预测效果却不好，这种情况称为**过拟合**（Over-fitting）。

通过大量参数，在训练数据上拟合复杂模型，可能很好地学习到训练数据集上的细微特征，也容易捕捉到不具一般性的样本噪声，过拟合为模型引入“偏见”，增加了模型的泛化误差。

3.4.2 正则化

一种过拟合的缓解方法，是根据模型的复杂程度，增加**罚项**（penalty term），使模型的结构风险最小化。

$$\text{Loss\_total}(\boldsymbol{W}) = \frac{1}{N}\sum_{i=1}^{N}\text{Loss}(x_i; W) + \frac{\lambda}{2}\|W\|^2 \tag{3.3}$$

总损失中增加的罚项部分，也称为 L_2 **正则化项**（regularizer），其中，λ 是根据模型复杂度设定的罚项系数；乘数 $\frac{1}{2}$ 用于简化计算；$||\boldsymbol{W}||$ 是权值参数 $\boldsymbol{W}$ 的 L_2 范数。

3.4.3 正则化的效果

L_2 正则化的直观效果是：含有相对大分量的权值参数其罚项也越大；各分量相对小而分散的权值参数罚项也较小。

例如，输入向量 $\boldsymbol{x} = [1, 1, 1, 1]$ 和两个权值向量，$\boldsymbol{w}_1 = [0.25, 0.25, 0.25, 0.25]$，$\boldsymbol{w}_2 = [1, 0, 0, 0]$ 做内积运算后的结果都是 1。

$$\boldsymbol{w}_1 \cdot \boldsymbol{x} = (0.25 \times 1) \times 4 = 1$$

$$\boldsymbol{w}_2 \cdot \boldsymbol{x} = 1 \times 1 + 0 = 1$$

然而两个向量的罚项则相去甚远，由于 $\boldsymbol{w}_1$ 各分量相对小而分散，罚项只有 0.25，而 $\boldsymbol{w}_2$ 的 L_2 罚项是前者的四倍。

$$\|\boldsymbol{w}_1\|^2 = 0.25^2 \times 4 = 0.25$$
$$\|\boldsymbol{w}_2\|^2 = 1^2 + 0^2 \times 3 = 1$$

注：本例来自 Stanford University cs231n, Linear Classification 单元。

L_2 正则化后的模型，倾向于均衡评估输入样本点上全部各个维度的特征，而不是少数大分量值特征对分类结果的影响，来提高模型在测试数据集上的泛化（generalization）能力，缓解过拟合风险。

偏置项 $\boldsymbol{b}$ 是否也需要做正则化处理呢？

通常不需要。

上面的例子可以看到：输入样本点各个维度特征分量的数值特征，影响了内积计算结果；而偏置 $\boldsymbol{b}$ 既不参与内积计算，也不对特征分量做数值倍数放大，实践中用正则化方法处理偏置 $\boldsymbol{b}$，对模型效果提升不大，所以通常只对权值参数 $\boldsymbol{W}$ 做正则化处理。

3.5　第三个挑战：非线性可分问题

模型经过多个隐藏层处理，再传递到输出层；隐藏层中的节点，表达了从输入特征中抽取得到的更丰富特征；单层网络结构成为多层结构后，线性模型也成为线性组合模型，从而能处理图3.2所示的**异或**类样本的分类识别，适应更多线性分类场景。

然而，任意线性组合构成的模型，仍然是线性的，无法很好地处理非线性可分场景，参见图 3.5所示的非线性可分问题。

非线性可分问题的解决办法，是引入非线性函数，对原始输出做函数处理；如果把线性模型中间层中的节点 $\boldsymbol{h}$ 看成神经元，把这些神经元的输出，经过非线性函数处理，使之去线性化，即可使模型得以处理非线性可分的场景。这些处理神经元输出的非线性函数，也称为**激活函数（activation function）**。

3.6　激活函数

神经网络常被用来和大脑的生物神经元系统做类比；然而人类的神经系统有超过 860 亿个神经元，这些神经元通过神经突触互相连接，神经突触的数量

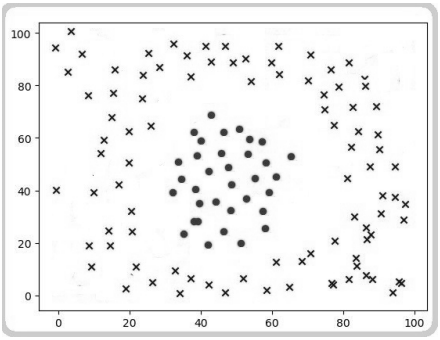

图 3.5　非线性可分问题

超过十万亿个，这些突触是稀疏连接的，连接之间的信号传输大部分时刻是被抑制的，只有少量超过阈值的信号，才会激活神经突触之间的连接，使信号得以被传递到网络的其他节点。在人工神经网络中，激活函数对节点输出起到类似的阈值控制作用。

图3.6列举了常见的激活函数的图像，神经网络的节点经过这些函数激活后，输出不再是输入的线性变换，模型也成为非线性神经网络模型，可以处理更复杂的非线性场景。

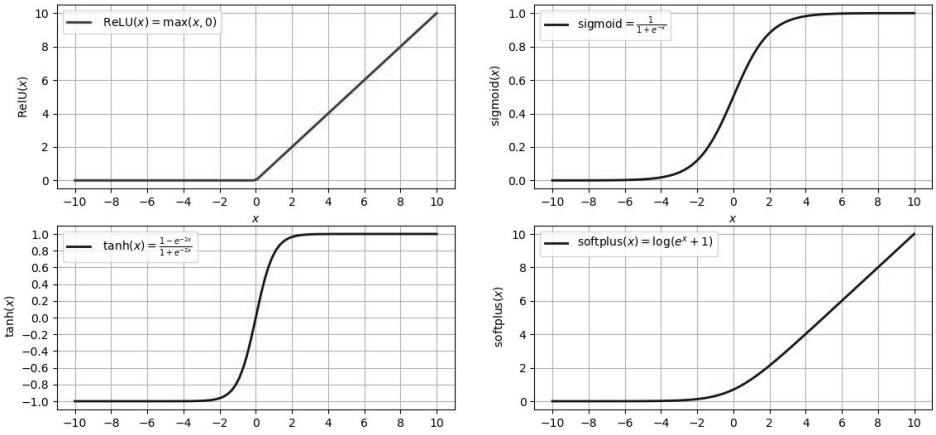

图 3.6　激活函数的图像，
上左：ReLU，上右：sigmoid
下左：tanh，下右：softplus

sigmoid 激活函数，也称为 S 型函数，把原始输出转换到 0 和 1 之间，在输出值很小的时候完全抑制，输出值很大时完全激活，二者之间的区域是连续可导的，易于计算。sigmoid 的不足之处是，位于输入值很大和很小的广阔区

域（称为饱和区），激活函数的输出几乎不再变化了，模型在这些区域难以学习到输入的特征，造成回传梯度过小的问题。这个问题在后续章节会结合反向传播算法做进一步介绍。

tanh 激活函数，也称为双曲正切函数，把原始输出转换到-1 和 1 之间，tanh 的输出数值以 0 为中心，这个特性可以使反向传播过程中参数的梯度更新，更平滑；同 sigmoid 激活函数类似，tanh 函数图像两侧也有广阔的饱和区，同样会带来回传梯度过小问题。

ReLU（Rectified Linear Units）激活函数，也称为线性整流函数，如图3.6所示，ReLU 函数有鲜明的特点：

- 稀疏的激活性；
- 宽阔的激活边界；
- 单侧抑制，仅激活输出大于阈值 0 的信号。

在深层神经网络中，ReLU 较前两种激活函数，降低信号激活率的作用较为明显，这个作用有助于更高效地提取重要特征；由于 ReLU 的计算逻辑更简洁，只需要把输出结果元素中的负数值置为 0 即可，能明显加快模型收敛。ReLU 激活函数的缺点是，如果学习率设置不当，可能在模型中产生大量失活节点，这种情况下，需要尝试给模型设置较小的学习率进行训练。

为了应对第三个挑战：使模型去线性化，我们在模型里增加 ReLU 激活，并调低学习率参数，来处理非线性可分问题。

从图3.6可以看到，ReLU 函数不是处处可导的，但是反向传播的梯度仍然可以计算，接下来会在算法部分介绍。

3.7 算法和结构

带着上述问题，我们继续了解这个多层神经网络的算法和整体结构。由于增加了正则化罚项损失部分，模型的总损失是：

$$\begin{aligned}\text{Loss\_total}(\boldsymbol{W}) &= \frac{1}{N}\sum_{i=1}^{N}\text{Loss}(\boldsymbol{x}_i;\boldsymbol{W}) + \frac{\lambda}{2}\|\boldsymbol{W}\|^2 \\ &= \frac{1}{N}\sum_{i=1}^{N}\text{Loss}(\boldsymbol{x}_i;\boldsymbol{W}) + \frac{\lambda}{2}\sum_i\sum_j \boldsymbol{W}_{ij}^2\end{aligned} \tag{3.4}$$

反向传播时，权值矩阵 $\boldsymbol{W}$ 也相应增加了正则化部分梯度：

$$\nabla \boldsymbol{W}_{\text{reg}} = \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{W}}(\frac{1}{2}\lambda \boldsymbol{W}^2) = \lambda \tag{3.5}$$

在输入层和输出层之间增加隐藏层之后，隐藏层的 $\boldsymbol{M}$ 个节点和输入层节点之间，有了新的权值矩阵 $\boldsymbol{W}_1$ 和偏置 $\boldsymbol{b}_1$：

$$\boldsymbol{h} = \text{ReLU}(\boldsymbol{x} \cdot \boldsymbol{W}_{1_{D\times M}} + \boldsymbol{b}_1) = \text{ReLU}(\boldsymbol{z}^h) \tag{3.6}$$

隐藏层的原始输出 $\boldsymbol{z}^h$ 仍然使用 ReLU 函数激活，激活后的输出表达为 $\boldsymbol{h}$，继续向前传递作为下一层的输入。

隐藏层到输出层之间连接的权重参数也需要训练，将其命名为 $\boldsymbol{W}_2, \boldsymbol{b}_2$，与前一层连接的权重参数对应并方便区分，图3.7所示是增加隐藏层之后的模型结构，训练启动前先对 $\boldsymbol{W}_1, \boldsymbol{b}_1$ 和 $\boldsymbol{W}_2, \boldsymbol{b}_2$ 做初始化处理。

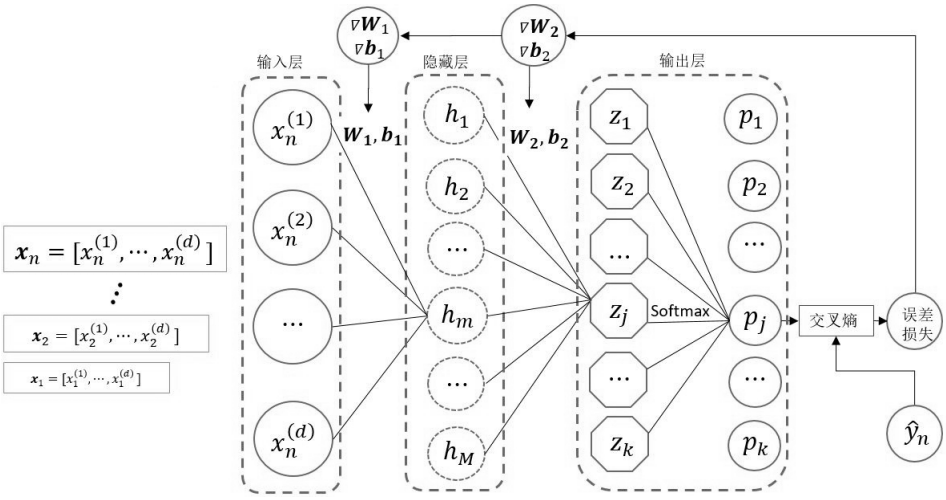

图 3.7　加入隐藏层的模型结构，模型有了两组训练参数，
反向传播误差从后向前依次传递，
隐藏层与前后相邻两层全部节点两两互相连接

隐藏层到输出层参数梯度计算方法不变，以隐藏层含有 M 个元素的输出 $\boldsymbol{h}$，作为后一层输入，再做一次仿射变换：

$$\boldsymbol{z} = \boldsymbol{h} \cdot \boldsymbol{W}_{2_{M\times K}} + \boldsymbol{b}_2 \tag{3.7}$$

仍然采用交叉熵计算误差损失，$\boldsymbol{W}_2, \boldsymbol{b}_2$ 反向传播梯度，正则化后，成为：

$$\begin{aligned} \nabla \boldsymbol{W}_2 &= (p - p_{\hat{y}}) \cdot \boldsymbol{h}^{\mathrm{T}} + \lambda \boldsymbol{W}_2 \\ \nabla \boldsymbol{b}_2 &= \sum_{j=1}^{K} (p - p_{\hat{y}}) \\ p_{\hat{y}_j} &= \begin{cases} 1 & , \ j = k \\ 0 & , \ j \neq k \end{cases} \end{aligned} \tag{3.8}$$

其中 $p_{\hat{y}}$ 的分量 $p_{\hat{y}_j}$ 的值，由 j 是否等于正确类别 k 而定。

对 $\boldsymbol{W}_1, \boldsymbol{b}_1$ 同样用梯度下降法（Gradient Descent），以反向传播得到梯度 $\nabla \boldsymbol{W}_1, \nabla \boldsymbol{b}_1$ 迭代更新参数来极小化误差损失。

首先，隐藏层本身没有损失函数，需要用下一处理层回传的误差损失，即本层的输出误差来计算参数梯度，后一层误差损失 $\boldsymbol{\delta}$ 反向传播到隐藏层：

$$\nabla h = \frac{\partial \mathrm{Loss}(\boldsymbol{W}_2, \boldsymbol{b}_2, \boldsymbol{z}, \hat{y})}{\partial \boldsymbol{h}} = \frac{\partial \mathrm{Loss}(\boldsymbol{W}_2, \boldsymbol{b}_2, \boldsymbol{z}, \hat{y})}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{h}} = \boldsymbol{\delta} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{h}} \tag{3.9}$$

又有

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{h}} = \frac{\partial [(\boldsymbol{W}_2 \cdot \boldsymbol{h} + \boldsymbol{b}_2]}{\partial \boldsymbol{h}} = \boldsymbol{W}_2^{\mathrm{T}} \tag{3.10}$$

所以

$$\nabla \boldsymbol{h} = (p - p_{\hat{y}}) \cdot \boldsymbol{W}_2^{\mathrm{T}} \tag{3.11}$$

若以标量 x 作为自变量，看激活函数本身 $\mathrm{ReLU}(x) = \max(0, x)$，不是处处可导的，但是观察函数性质，当自变量为负数或者 0 时，经过 ReLU 函数之后被抑制，其输出降为 0；如果自变量为正数则导数为 1，即误差传递到此处是自变量本身。

$$\frac{\mathrm{dRelU}(x)}{\mathrm{d}x} = \begin{cases} 1 & , \ x > 0 \\ 0 & , \ x \leqslant 0 \end{cases} \tag{3.12}$$

所以实际反向梯度 $\nabla \boldsymbol{h}$ 在位置 i 处的分量 ∇h_i 为

$$\nabla h_i = \begin{cases} \nabla h_i & , \ h_i > 0 \\ 0 & , \ h_i \leqslant 0 \end{cases} \tag{3.13}$$

添加正则化处理后，参数更新的梯度:

$$\begin{aligned} \nabla \boldsymbol{W}_1 &= \boldsymbol{x}^{\mathrm{T}} \cdot \nabla \boldsymbol{h} + \lambda \boldsymbol{W}_1 \\ \nabla \boldsymbol{b}_1 &= \sum_{m=1}^{M} \nabla \boldsymbol{h} \qquad \blacksquare \end{aligned} \tag{3.14}$$

至此，得到反向传播的全部参数更新梯度，每完成一次迭代训练，即可使用计算得到的梯度，更新各个处理层的权值和偏置项参数。

3.8 算法实现

3.8.1 数据准备

这个案例不需要再做特别的数据准备，可以复用第 2 章实现的 MNIST 数据处理类，得到已随机分组的 mini-batch 批样本和验证数据集，分别用于迭代训练和推理验证。

3.8.2 实现多层全连接神经网络

和上一章的单层模型相比，这一次的模型增加了隐藏层，并使用 ReLU 函数对原始输出做激活处理：

$$\boldsymbol{h} = \mathrm{ReLU}(\boldsymbol{x} \cdot \boldsymbol{W}_1 + \boldsymbol{b}_1)$$

ReLU 激活函数对该层神经元的原始输出做单侧抑制：

```
1 hidden_layer = np.maximum(0, np.matmul(x, w) + b)
```

对隐藏层输出再做一次变换：

$$\boldsymbol{z} = \boldsymbol{h} \cdot \boldsymbol{W}_2 + \boldsymbol{b}_2$$

```
z = np.matmul(hidden_layer, w2) + b2
```

接下来，对输出做 Softmax 处理，并计算交叉熵损失 LossCE，处理逻辑仍然和单层模型一样，可参见第 2 章的实现部分。

为模型引入 L_2 正则化处理后，在交叉熵损失上，增加正则化罚项：

$$\begin{aligned}\text{Loss\_total}(\boldsymbol{W}) =& \frac{1}{N}\sum_{i=1}^{N}\text{Loss}(x_i;\boldsymbol{W}) + \frac{\lambda}{2}\|\boldsymbol{W}\|^2 \\ =& \frac{1}{N}\sum_{i=1}^{N}\text{Loss}(x_i;\boldsymbol{W}) + \frac{\lambda}{2}\sum_i\sum_j \boldsymbol{W}_{ij}^2\end{aligned}$$

正则化罚项的系数 lambda 设置一个较小的值，这个例子中，设置为 0.0001：

```
# 正则化罚项系数lambda
LAMDA = 1e-4
# 计算正则化损失罚项
reg_loss = REG_PARA * LAMDA * (np.sum(w*w) + np.sum(w2*w2)
    )
# 总损失增加正则化部分
loss = lossCE+regloss
```

正则化罚项只在训练时参与前向计算，参数训练完成后，需要跳过正则化逻辑进行推理预测。反向传播时，权值矩阵 $\boldsymbol{W}_2$ 也相应地增加了正则化部分梯度：

$$\nabla \boldsymbol{W}_{\text{reg}} = \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{W}}(\frac{1}{2}\lambda\|\boldsymbol{W}\|^2) = \lambda\boldsymbol{W}$$

```
# 反向传播的原始参数梯度
delta_w2 = np.dot(hidden_layer.T, delta_y_mean)
# 正则化部分梯度
delta_w2 += LAMDA * w2
```

误差继续反向传播到隐藏层的激活环节：

$$\nabla \boldsymbol{h} = (p - p_{\hat{y}}) \cdot \boldsymbol{W}_2^{\mathrm{T}}$$

```
# 误差向前层传递
dhidden = np.dot(delta_y_mean, w2.T)
```

激活函数 ReLU，前向输出元素为负数和 0 的位置，经 ReLU 之后被抑制，其反向传播梯度降为 0；如果前向输出为正数，则反向传播时导数为 1，即误差传递到此处是自变量本身, 所以实际的梯度为

```
# 非线性激活函数ReLU的反向传播
dhidden[hidden_layer <= 0] = 0
```

观察激活函数 ReLU 的实现，其前向计算和反向传播都非常简洁。与其他常见激活函数相比，简单而有效也是 ReLU 函数的一大优势。

隐藏层的权参 $\boldsymbol{W}_1$ 同样增加正则化部分梯度。

$$\nabla \boldsymbol{W}_1 = \boldsymbol{x}^{\mathrm{T}} \cdot \nabla \boldsymbol{h} + \lambda \boldsymbol{W}_1$$

```
# 首层参数的原始反向传播梯度
delta_w1 = np.dot(x.T, dhidden)
# 正则化部分梯度
delta_w1 += LAMDA * w1
```

每次迭代训练后和单层网络是一样的，用梯度下降方法更新参数，需要用同样的逻辑对两组参数分别做更新。

```
# 更新w,b参数
w = w - LEARNING_RATE * delta_w
b = b - LEARNING_RATE * delta_b
```

上述算法的 Python 实现，**不借助深度学习框架**，在全连接神经网络的基本结构上，增加了正则化处理，缓解过拟合问题，并添加了一个隐藏层，再引入激活函数，使模型能处理异或问题，识别非线性可分特征，进一步提高分类正确率。

3.8.3 在数据集上验证模型

以下是设置单隐藏层 512 个隐藏层节点的训练过程，模型在验证数据集上，能够稳定收敛, 如图3.8所示。

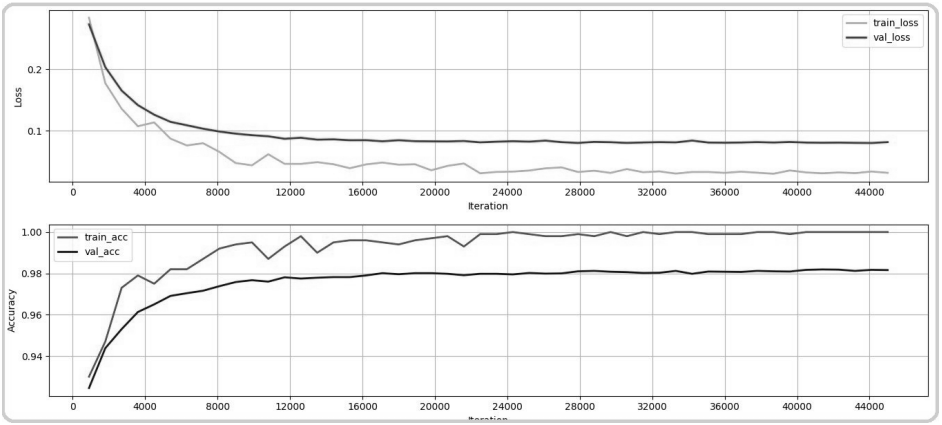

图 3.8 多层全连接神经网络的指标曲线，添加了一个隐藏层后，在独立验证集的正确率有了明显提高

和单层网络相比，手写数字识别的正确率，有了大幅提高。经过迭代训练，在验证集上的正确率稳定在 98% 以上。

```
start..
w,b init..
learning_rate= 0.10000000
epoch  0, loss_v=1.228955, acc_v=0.740500
epoch  1, loss_v=0.152158, acc_v=0.948500
epoch  2, loss_v=0.126728, acc_v=0.959000
epoch  3, loss_v=0.113179, acc_v=0.968000
epoch  4, loss_v=0.103391, acc_v=0.970500
epoch  4, loss_v=0.112588, acc_v=0.967000
epoch  5, loss_v=0.102079, acc_v=0.973500
epoch  5, loss_v=0.101361, acc_v=0.970500
epoch  6, loss_v=0.094584, acc_v=0.970500
epoch  7, loss_v=0.084534, acc_v=0.974500
epoch  8, loss_v=0.070888, acc_v=0.982500
...
Training and validation end.
```

只有一个隐藏层的神经网络，在数据集上的表现已经超过采用径向基核函数的支持向量机模型（RBF-SVM）。

3.9 小结

这一章介绍了具有一个隐藏层的全连接神经网络，随着深度（隐藏层）增加，神经网络能在复杂的输入样本上提取更多的特征，分类效果超过支持向量机模型，在一些数据集上，甚至超过了人工识别正确率。

增加深度也带来了新挑战：连接节点和参数过多，导致反向传播训练效率降低，且增加了过拟合风险。下一章，我们将在目前网络结构的基础上，引入卷积核（Convolutional Kernel）处理，使模型进化成为更强大的卷积神经网络（CNN），通过进一步实践来应对这一挑战。

参 考 文 献

[1] Y LeCun, L Bottou, Y Bengio, P Haffner. **Gradient-based learning applied to document recognition.** Proceedings of the IEEE, 1998, 86(11):2278-2324.

[2] Dan Jurafsky, James H Martin. **Speech and Language Processing**. Prentice Hall, 2000.

[3] Jürgen Mayer, Khaled Khairy, Jonathon Howard. **Drawing an elephant with four complex parameters**. American Journal of Physics, 2010.

[4] Alexander LeNail. **NN-SVG: Publication-Ready Neural Network Architecture Schematics**. Journal of Open Source Software, 2019, 4(33):747.

第 4 章

卷积神经网络（CNN）

“归山深浅去，须尽丘壑美。
莫学武陵人，暂游桃源里。”

——[唐] 裴迪《送崔九》

“卷积神经网络”是公开数据集上，历届夺冠算法的核心支撑模型。

上一章，**不借助深度学习框架**，在神经网络的基本结构上，增加了**隐藏层**并**激活**，得以提取非线性可分的高层特征，并介绍了一种过拟合的缓解方法，使模型在 MNIST 的验证数据集上得到大于 98% 的预测正确率。这一章，继续为这个模型引入卷积处理，使模型进化为**卷积神经网络**（Convolutional Neural Networks，CNN），通过抽取更丰富的高层特征，实现更精准的分类识别。

4.1 挑战：参数量和训练成本

引入隐藏层，使模型通过抽取高层特征，具备了处理非线性可分样本的能力，也带来新的挑战：一个具有 784 维特征的样本，网络中每增加一层 500 个节点的隐藏层，模型需要引入 $784 \times 500 = 392{,}000$ 个权值 $\boldsymbol{W}$ 参数项和 500 个偏置 $\boldsymbol{b}$ 参数项，对彩色或更大尺寸的输入图片，则需要引入更多参数，参数量增加提高了模型的过拟合风险，也带来更大计算量，使更多层（更深）全连接神经网络的训练成本上升，直至不再可行。

卷积神经网络是应对这一挑战的有效方法。

4.2 卷积神经网络的结构

卷积神经网络在模型中引入了新的结构：卷积层（convolutional layer）和池化层（pooling layer）。

网络模型由各个处理层叠加构成，可以用正则表达式描述如下：

输入层 ->{{卷积层}+ ->{池化层}？ } + ->{全连接层}+ -> 输出层

在输入层之后，是一个或多个卷积层，每个卷积层可以单独作为一层；也可以让一个卷积层搭配一个池化层，组成 Conv-Pool 单元；多个卷积层或者 Conv-Pool 单元串联（如 LeNet）, 或并联（如 GoogLeNet）组合在一起，置于输入层和全连接层（FC Layer）之间，一起构成整个卷积神经网络。

图4.1所示的 CNN 结构，全部由卷积层和全连接层构成，每一次卷积处理都可以改变输入数据的宽高尺寸和深度，经过卷积处理后，通常宽高尺寸减小而深度增加。

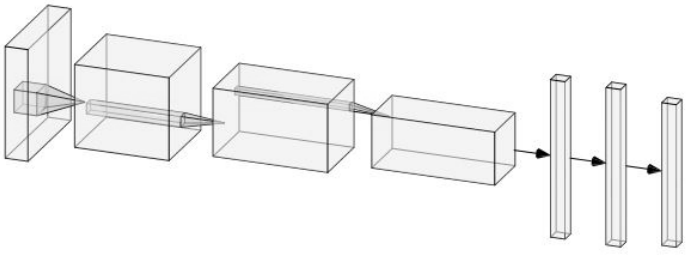

图 4.1　由卷积层和全连接层构成的网络，立方体代表的是数据而不是处理层。左边的立方体代表输入数据，中间的三个立方体代表经过卷积处理后的数据，右侧的三个立方柱分别代表三个全连接层的输出

在卷积神经网络的结构里，输入层的图片数据不再是一阶**向量**（vector），

而是 n 阶**张量**（tensor），一张 RGB 色彩模式的图片可以看成通道（channel）为 3 的张量，每个深度上的矩阵元素，表达当前颜色通道下不同位置的亮度。

与全连接层不同，卷积层和池化层上的每个节点，分别只与相邻前一层上，一小部分区域内的节点相连。卷积层与池化层，本质上是在本层输入和输出的三维数据之间，用可微分函数（differentiable function）做转换处理。卷积层函数带有需要训练的参数，而池化层的处理函数通常没有参数需要训练。

4.2.1 卷积层

> "Convolutional neural nets are based on the simple fact that a vision system needs to use the same knowledge at all locations in the image."
>
> "卷积神经网络立足于一个简单的事实：视觉系统在图像的不同区域上使用了相同的理解机制。"
>
> Geoffrey Hinton,2018

相对于全连接神经网络，卷积神经网络结构之所以能够有效地减少模型需要训练的参数总量，原因是卷积层的权值共享机制。

权值共享

某一个特征在图片的不同位置是同等重要的，基于这个假设前提，卷积层用一套共享的权值参数："**过滤器（Filter）**"来抽取图像所有通道的高层特征，将输入数据转换为下一层的输出，CNN 中的卷积过滤器，也被称为 "**卷积核（kernel）**"。过滤器的个数也被称为卷积核的**深度（depth）**。图 4.2所示卷积层计算的例子中，过滤器个数（depth）为 1，过滤器权值参数 $\boldsymbol{W}$ 有三个分量矩阵，分别与输入张量 $\boldsymbol{x}$ 的 3 个通道（channel）矩阵做运算。通常一个卷积层会有多个过滤器，每个过滤器都有和输入通道数量对应的权值参数分量矩阵，各自完成图 4.2所示的处理步骤，得到一个输出深度上的分量矩阵。

在每个输入通道的矩阵上，依次用过滤器同尺寸 $F_{\text{height}} \times F_{\text{width}}$ 的数据块（block）, 与对应的 $\boldsymbol{W}$ 分量矩阵分别做点积运算，在图 4.2所示这个例子中，3 组分量矩阵分别做点积运算，得到的标量结果, 再与偏置项 $\boldsymbol{b}$ 对应深度的分量相加，仍得到一个标量，作为输出矩阵 Out 同深度上对应位置的标量元素值。

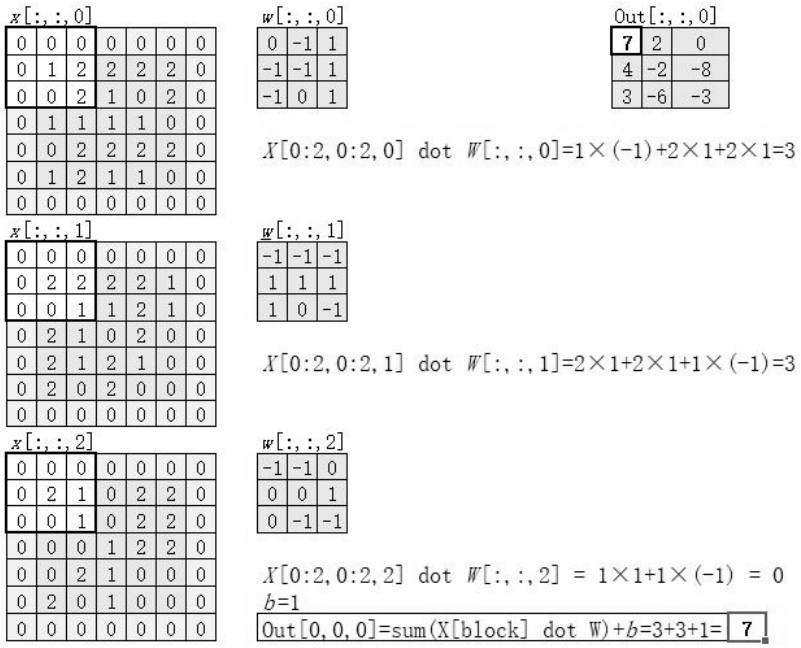

图 4.2　卷积层计算——卷积核滑动, 第一步

在输入矩阵上，从左到右自上而下，顺序滑动过滤器，选取数据块（block）完成上述运算，得到输出矩阵 Out 的全部元素。参见图 4.3, 卷积核向右滑动，步长为 2，进行相应卷积计算，得到当前深度上输出矩阵对应位置的一个元素。各个深度上的卷积核依次和每个通道上的输入数据做点积与求和运算，最终得到所有输出深度上的结果矩阵。经过卷积运算，输入数据的通道维度，变成了与卷积核个数一致的深度维度，每个深度上的矩阵宽高尺寸，也发生了变化。

步长（strides）

在输入矩阵上选取输入数据块（block）时，可以在宽（width）和高（height）两个方向上逐元素**“顺序移动”**，即步长为 strides = 1；也可以用特定**步长**（strides>1）**“跳跃移动”**，压缩输出矩阵的尺寸，换取计算速度提升，在图 4.2和图 4.3所示的例子中，步长 strides = 2。

在步长 strides ⩾ 2 的情况下，可能会出现输入矩阵的剩余元素小于卷积核尺寸，从而不足以完成卷积运算的情况，此时，如果不做特殊处理，这部分输入元素将无法参与卷积运算中，这部分数据所表达的输入特征也无法被卷积运算所捕获，为了确保所有输入元素不遗漏地参与计算，在处理之前，可能需

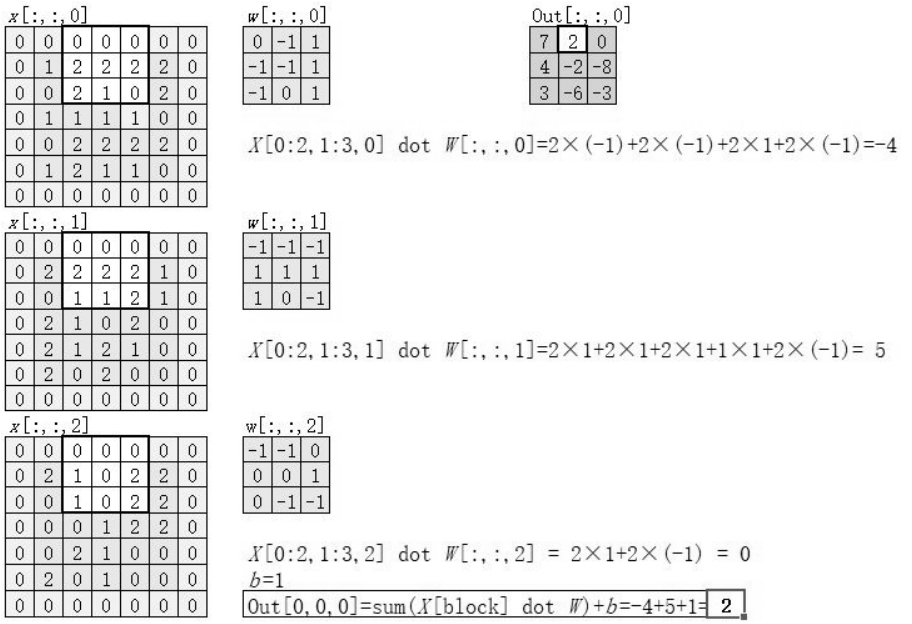

图 4.3　卷积层计算——卷积核滑动，第二步

要先对输入矩阵做边缘填充（padding）。

边缘填充

边缘填充可以像图 4.4 的例子一样，使用全零填充（zero-padding），也可以使用就近元素值来填充。

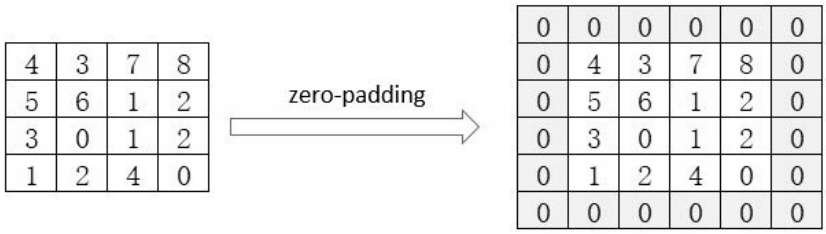

图 4.4　全零填充

是否需要填充，以及填充的尺寸 padding ，可以用下面的公式判断和推算：

$$\text{padding} = [(\text{size}_{\text{output}} - 1)/\text{strides} + \text{size}_{\text{filter}} - \text{size}_{\text{input}}]/2 \qquad (4.1)$$

例如：

当输入矩阵为 7×7, 卷积核为 3×3，步长 strides=1, 输出尺寸为 5×5 时，**padding = 0**, 不需要做填充。

当输入矩阵为 3×3, 卷积核为 2×2，步长 strides=1, 输出尺寸为 4×4 时, **padding = 1**，需要做一层边缘填充。

同样的，如果已经知道输入矩阵宽高尺寸 size_i 和卷积核尺寸 $\text{size}_\text{filter}$，以及填充尺寸 padding、卷积步长 strides，也可以用下面的公式推算卷积输出矩阵的宽高尺寸：

$$\text{size}_\text{o} = (\text{size}_\text{i} - \text{size}_\text{filter} + 2 \times \text{padding})/\text{strides} + 1 \tag{4.2}$$

例如：

当输入矩阵为 9×9, 卷积核为 3×3，步长 strides=2, 填充 padding=1 时，卷积输出的宽高尺寸为 $(9 - 3 + 2 \times 1)/2 + 1 = 5$。

当输入矩阵为 227×227, 卷积核为 11×11，步长 strides=2, 填充 padding=0 时，卷积输出的宽高尺寸为 $(227 - 11 + 2 \times 0)/4 + 1 = 55$。

实践中，如果需要根据数据场景构造自己的卷积神经网络模型，可以参考式 (4.1) 和式 (4.2) 来设计卷积层的超参数。

多个卷积核

为了抽取更丰富的特征，一个卷积层往往有多个过滤器，即 $F_\text{depth} > 1$，此时，输出结果 O 会有 F_depth 个分量矩阵，或者说卷积层的输出深度为 $D = F_\text{depth}$ 。

输入数据与过滤器经过上述卷积运算后，在输出矩阵某个输出深度 d 上，位于第 i 行 j 列的元素值为

$$z_{d,i,j} = \sum_{\text{ch}=0}^{\text{CH}-1} \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} w_{d,\text{ch},r,c} x_{\text{ch},i+r,j+c} + b_d \tag{4.3}$$

其中 ch 为输入数据通道（channel），r 和 c 分别为过滤器在高度和宽度上的尺寸。

运算结果经过 ReLU 激活函数，得到的当前卷积层在深度 d 上激活后的

输出：

$$a_{d,i,j} = \mathrm{ReLU}(z_{d,i,j}) \tag{4.4}$$

多个输出深度上的矩阵，共同构成当前卷积层输出 a，同时也作为 CNN 下一层的输入。

卷积神经网络所指的卷积运算，与数学意义上卷积（本书中记作 $\circledcirc$）的定义不同，CNN 中的卷积运算本质上是互相关（cross-correlation）运算（本书中记作 $\odot$）；如果没有特别说明，接下来章节中的**卷积**、**互相关**都是指神经网络中的卷积 $\odot$，用**经典卷积**的提法指代数学上的卷积运算 $\circledcirc$。

卷积神经网络中，卷积核参数是经过训练得到的，因此用于 CNN 模型参数训练时，只要前向计算和反向传播的计算规则保持一致，经典卷积与互相关这两种运算对于模型的训练结果是等效的。

你可以进一步了解经典卷积运算和互相关运算的差异，也可以跳过这一节，从**池化层**继续阅读，不影响对卷积神经网络的整体理解和实现。

经典卷积与互相关

数学意义上的经典卷积运算是通过两个函数 f 和 g 生成第三个函数 S 的运算，S 表征函数 f 与经过翻转和平移的函数 g 的乘积的积分。

经典卷积运算的一阶连续形式：

$$S(x) = \int_{-\infty}^{\infty} f(\tau)g(x-\tau)\mathrm{d}\tau \tag{4.5}$$

回忆积分的几何意义，它是乘积函数 S 所围成的曲边梯形的面积。

二阶离散形式：

$$\mathrm{Conv}_{i,j} = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} w_{m,n}x_{i-m,j-n} \tag{4.6}$$

对上一节的式 (4.3)，如果不考虑输入通道（channel）这个维度与输出深度 d，再拿掉偏置项 $\boldsymbol{b}$，则可以简化为式 (4.7)：

$$\text{CrossCorr}_{i,j} = \sum_{r=0}^{R-1}\sum_{c=0}^{C-1} w_{r,c}x_{i+r,j+c} \tag{4.7}$$

比较经典卷积（convolution）运算和互相关（cross-correlation）运算的过程，会发现这两种运算仅是运算符号有差异，这个差异使得输入数据 $\boldsymbol{x}$ 与过滤器权值矩阵 $\boldsymbol{W}$ 做点积运算时，$\boldsymbol{x}$ 数据块（block）中的元素与 $\boldsymbol{W}$ 中的元素，在经典卷积运算时是逆序对应的，因此如果把经典卷积核参数矩阵 $\boldsymbol{W}$ 旋转 180° 作为过滤器，经典卷积运算就成了互相关运算。

这个差异使经典卷积运算满足了结合律（Associative Law）：

$$Fa \circledcirc (Fb \circledcirc G) = (Fa \circledcirc Fb) \circledcirc G \tag{4.8}$$

式 (4.8) 中的 $\circledcirc$ 运算符，代表数学意义上的经典卷积运算。满足结合率这个性质，在需要连续多次经典卷积的场景下非常实用：你可以先把各个经典卷积核依次做经典卷积运算，得到新的**叠加经典卷积核**，再用叠加经典卷积核与输入数据做一次经典卷积得到最终结果；如果这些经典卷积核的权值参数都是已知常数，则可以更进一步：预先计算**叠加经典卷积核**并存储起来直接复用，成倍提高运算效率。

互相关运算不满足结合律，目前成熟的 CNN 模型也不需要做多个过滤器直接叠加的互相关运算；互相关运算另有效果拔群的优化方法，接下来的章节会做具体介绍。

通常，互相关运算用于图像模式匹配（template matching），经典卷积运算用于图像柔化处理（smoothing）。CNN 模型训练的时候，参数不是预先固定的常量，而是通过反向传播误差损失、迭代优化得到的，所以前向传播时过滤器 $\boldsymbol{W}$ 无论翻转与否，对模型参数训练是等效的。

4.2.2 池化层

池化层改变输入张量中各个分量矩阵高与宽的尺寸，仅缩小矩阵的大小，不改变三维张量的深度；池化层减少全连接层中的节点个数，也通过降低整个神经网络的参数量缓解过拟合风险。

池化处理也称为降采样（sub-sampling），可以直观理解为，降低了图片

的分辨率，同时保留图像的一部分特征。

池化处理的做法，同样是采用固定尺寸的过滤器，在输入矩阵上依次移动，摘取对应数据块像素上的最大值（max-pooling）或平均值（mean-pooling），构成输出矩阵的元素，作为采样结果；两种池化运算，分别被称为最大池化和平均池化运算。

例如：图4.5所示的最大池化计算，采用尺寸为 2×2 的过滤器，对所有深度上的节点矩阵，以步长 strides = 2，在高与宽两个维度跳跃移动，摘取 4 个元素数据块上的最大值作为采样结果，从而将节点矩阵尺寸压缩到原来的四分之一。

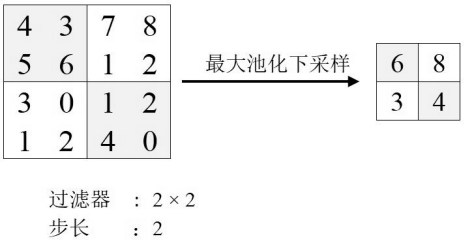

图 4.5 最大池化计算

若 size_i 为池化层输入矩阵尺寸，F 为池化过滤器尺寸，S 为步长 strides，则可以推算输出矩阵的尺寸 size_o。

$$\text{size}_o = (\text{size}_i - F)/S + 1 \tag{4.9}$$

4.2.3 全连接层和 Softmax 处理

经过卷积层和池化层处理，抽取得到更高层特征后，经过若干全连接层完成分类，最后由 Softmax 函数将分类结果转换为概率分布。

全连接层与 Softmax 函数的处理逻辑可参考之前的章节，接下来重点介绍 CNN 的学习算法。

4.3 卷积神经网络学习算法

4.3.1 全连接层

首先，回顾**全连接层**对误差 $\boldsymbol{\delta}$ 的反向传播。

第 L 层（全连接层）原始输出：

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)} \cdot \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)} \tag{4.10}$$

该层经过函数 f 激活后的输出:

$$\boldsymbol{a}^{(l)} = f(\boldsymbol{z}^{(l)}) \tag{4.11}$$

第 L -1 层的原始输出误差 $\boldsymbol{\delta}$ 可以由 L 层的输出误差反推得到:

$$\boldsymbol{\delta}^{(l-1)} = \left((\boldsymbol{W}^{(l)})^{\mathrm{T}}\boldsymbol{\delta}_{fc}^{(l)}\right) \otimes f'(\boldsymbol{z}^{(l-1)}) \tag{4.12}$$

参数的梯度，由该层的**激活前输出误差**和上一层**激活后输出**推算：

$$\begin{aligned} \nabla_{\boldsymbol{W}^{(l)}}\mathrm{Loss}(\boldsymbol{W},\boldsymbol{b};\boldsymbol{x},y) &= \boldsymbol{\delta}^{(l)}(a^{(l-1)})^{\mathrm{T}} \\ \nabla_{\boldsymbol{b}^{(l)}}\mathrm{Loss}(\boldsymbol{W},\boldsymbol{b};\boldsymbol{x},y) &= \boldsymbol{\delta}^{(l)} \end{aligned} \tag{4.13}$$

池化层和卷积层由于节点间采取局部连接，误差传递不同于全连接层，下面分别来看。

4.3.2 池化层反向传播

池化层前向计算时，对本层输入数据移动过滤器定位数据块，逐数据块（block）地做获取最大值或平均值的固定处理，只有步长（strides）和过滤器尺寸（filter_size）超参数，没有权值参数需要训练，只需将误差 $\boldsymbol{\delta}$ 反向传播至上一层。这个过程是下采样的逆处理，称为**上采样**（up-sampling）。分别来看最大池化层和平均池化层的上采样过程：

做**最大池化层**（max-pooling）处理时，输出矩阵的每组数据块，只有 max-value 元素被采样，所以每个数据块中的 max-value 元素的梯度为 1，其他元素不参与误差传递，梯度为 0。为了让最大池化层的误差 $\boldsymbol{\delta}$ 回传时对号入座，在最大池化处理时，输入数据每个深度上各个数据块 max-value 的位置索引，需要在前向传播采样后保存下来，反向传播时直接复用，如图4.6所示。

平均池化层（mean-pooling）处理更简单，输出矩阵的每个数据块（block）中各元素的平均值被采样，所以误差 $\boldsymbol{\delta}$ 反向传播时，只要等权重平均分配到

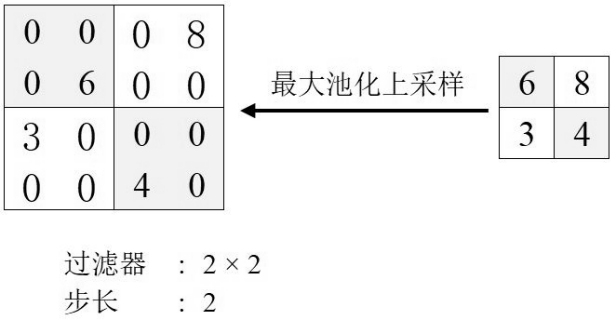

图 4.6 最大池化层反向传播: 上采样

误差矩阵各个数据块对应元素位置即可。

结合两种上采样处理，池化层对误差的反向传播可以表示为

$$\boldsymbol{\delta}^{(l-1)} = \text{upsample}(\boldsymbol{\delta}^{(l)} \otimes f^{'}(\boldsymbol{z}^{(l-1)}) \tag{4.14}$$

池化层误差继续反向传递来到卷积层。

4.3.3 卷积层反向传播

卷积层的反向传播算法,在宽与高两个维度上,可分为卷积步长 strides = 1 和卷积步长 strides > 1 两种情况。

当卷积的**步长 strides= 1** 时，在每一个输入通道 channel 上，一个过滤器（filter）的**卷积层误差 $\boldsymbol{\delta}$** 传递，按照链式法则展开：

$$\begin{aligned} \delta_{i,j}^{(l-1)} &= \frac{\partial \text{Loss}^{(l)}}{\partial z_{i,j}^{(l-1)}} \\ &= \frac{\partial \text{Loss}^{(l)}}{\partial a_{i,j}^{(l-1)}} \frac{\partial a_{i,j}^{(l-1)}}{\partial z_{i,j}^{(l-1)}} \end{aligned} \tag{4.15}$$

式 (4.15) 结果的第一部分 $\dfrac{\partial \text{Loss}^{(l)}}{\partial a_{i,j}^{(l-1)}}$，结合互相关（cross-correlation）的定义，对误差元素逐项求导可以观察到，上一层误差矩阵是本层误差矩阵先做边缘填

充（padding）后，再与旋转 180° 的过滤器（filter）做互相关运算的结果：

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & \delta_{11} & \delta_{12} & 0 \\ 0 & \delta_{21} & \delta_{22} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \odot \begin{pmatrix} w_{22} & w_{21} \\ w_{12} & w_{11} \end{pmatrix} = \begin{pmatrix} \nabla a_{11} & \nabla a_{12} & \nabla a_{13} \\ \nabla a_{21} & \nabla a_{22} & \nabla a_{23} \\ \nabla a_{31} & \nabla a_{32} & \nabla a_{33} \end{pmatrix} \tag{4.16}$$

说明：此处 $\odot$ 表示**互相关（cross-correlation）**运算。即

$$\frac{\partial \text{Loss}^{(l)}}{\partial a_{i,j}^{(l-1)}} = \boldsymbol{\delta}^{(l)} \odot \text{rot180°}(\boldsymbol{W}^{(l)}) \tag{4.17}$$

展开为逐项累加形式：

$$\frac{\partial \text{Loss}^{(l)}}{\partial a_{i,j}^{(l-1)}} = \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} w_{r,c}^{(l)} \delta_{i-r,j-c}^{(l)} \tag{4.18}$$

误差 $\boldsymbol{\delta}$ 的数据块元素与过滤器 $\boldsymbol{W}$ 的元素，逆序对应再做点积运算，这是数学上定义的经典卷积运算，与前向预测的互相关运算不同。

式 (4.15) 第二部分 $\dfrac{\partial a_{i,j}^{(l-1)}}{\partial z_{i,j}^{(l-1)}}$ 是激活函数 f 的导函数：

$$\frac{\partial a_{i,j}^{(l-1)}}{\partial z_{i,j}^{(l-1)}} = f'(z_{i,j}^{(l-1)}) \tag{4.19}$$

两部分合在一起：

$$\begin{aligned} \delta_{i,j}^{(l-1)} &= \frac{\partial \text{Loss}^{(l)}}{\partial z_{i,j}^{(l-1)}} \\ &= \frac{\partial \text{Loss}^{(l)}}{\partial a_{i,j}^{(l-1)}} \frac{\partial a_{i,j}^{(l-1)}}{\partial z_{i,j}^{(l-1)}} \\ &= \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} w_{r,c}^{(l)} \delta_{i-r,j-c}^{(l)} f'(z_{i,j}^{(l-1)}) \end{aligned} \tag{4.20}$$

式 (4.20) 写为互相关运算形式：

$$\boldsymbol{\delta}^{(l-1)} = \boldsymbol{\delta}^{(l)} \odot rot180°(\boldsymbol{W}^{(l)} \otimes f'(\boldsymbol{z}^{(l-1)}) \tag{4.21}$$

式 (4.21) 是 strides=1 时的情况。

当**步长 strides $>$ 1** 时，前向卷积结果较 strides=1 的运算差异，是每一步跳过了部分元素后再计算卷积输出，所以反向传播时，先扩展误差矩阵，补全跳过部分的元素位置，这部分数据不在网络中传递，因此直接在增补位置填梯度 0，问题就成了步长为 1 时的误差矩阵求解，仍然用式 (4.21) 计算上一层误差。

输入通道（channel）和过滤器（filter）个数（depth）都大于 1 时，把 D 层深度误差矩阵的每一层，逐个矩阵与该层的 C 个输入通道的本层输入矩阵做卷积，得到 C 个上一层误差矩阵分量，所有 D 层深度上的误差矩阵做同样运算，得到 D 组，每组 C 个，合计 $D \times C$ 个误差矩阵分量；在 D 这个维度上将各分量按元素相加，就合并成 C 个误差矩阵分量，这就是传导到上一层的误差张量 $\boldsymbol{\delta}$, 它的维度规格与多个通道的原始输入数据一致。

$$\boldsymbol{\delta}^{(l-1)} = \sum_{d=0}^{D} \boldsymbol{\delta}_d^{(l)} \odot rot180^o(\boldsymbol{W}_d^{(l)}) \otimes f'(\boldsymbol{z}^{(l-1)}) \tag{4.22}$$

计算卷积过滤器的权值参数梯度$\nabla \boldsymbol{W}$：

$$\frac{\partial \text{Loss}}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial \text{Loss}}{\partial \boldsymbol{z}^{(l)}} \frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{W}^{(l)}} \tag{4.23}$$

对式 (4.23) 的第二部分，由于 $\boldsymbol{z}$ 是以 $\boldsymbol{W}$ 作为过滤器，做互相关运算得出的：

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)} \odot \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)} \tag{4.24}$$

对式 (4.24) 运算逐项展开求导可以观察到，本层过滤器权值参数梯度，是以本层误差张量 $\boldsymbol{\delta}$ 作为过滤器和上一层激活后输出做互相关运算的结果：

$$\frac{\partial \text{Loss}^{(l)}}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial \text{Loss}^{(l)}}{\partial \boldsymbol{z}^{(l)}} \frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{a}^{(l-1)} \odot \boldsymbol{\delta}^{(l)} \tag{4.25}$$

式 (4.25) 展开为逐项累加形式：

$$\frac{\partial \text{Loss}^{(l)}}{\partial w_{i,j}^{(l)}} = \sum_{r=0}^{R-1}\sum_{c=0}^{C-1} \delta_{r,c}^{(l)} a_{i+r,j+c}^{(l-1)} \tag{4.26}$$

计算卷积过滤器的偏置梯度∇b：

有 D 个过滤器的卷积核，偏置项梯度同样有 D 个，顺位 d 的这个卷积核偏置项梯度，是该网络层误差张量 $\boldsymbol{\delta}$ 在 d 这个分量矩阵上各个元素的和：

$$\frac{\partial \text{Loss}^{(l)}}{\partial b_d^{(l)}} = \frac{\partial \text{Loss}^{(l)}}{\partial z_d^{(l)}} \frac{\partial z_d^{(l)}}{\partial b_d^{(l)}} = \sum_i \sum_j \delta_{d,i,j}^{(l)} \quad \blacksquare \tag{4.27}$$

以上，是 CNN 模型中全连接层以及新引入的卷积层、池化层反向传播的具体算法，也是支撑各大深层 CNN 模型的基础构件。

4.4 算法实现

4.4.1 数据准备

加入卷积处理后，神经网络模型对数据特征的捕获能力，有了大幅提升，为了直观地比较提升效果，仍然以 MNIST 数据集作为输入数据，使用卷积神经网络模型做分类训练和测试。

卷积神经网络的输入图片规格，与之前的数据准备方式有所不同：首先，图片不再被拉伸（reshape）为 784 维的向量，而是保持 28×28 像素的原始规格；此外，还需要增加一维颜色通道，处理彩色图片时，新增的颜色通道维度即输入数据的 channel = 3，用来表达 RGB 三种颜色通道上的亮度。

我们实现的卷积神经网络模型，本身可以支持颜色通道为 3 的彩色图片分类，由于在目标问题下，MNIST 数据是灰度图片，只有一个颜色通道，所以增加的一维深度设置为 1：

```
# MINIST是灰度图片，只有一个颜色通道，但仍增加通道维度
# 使模型支持多通道彩色图片处理
images = np.fromfile(imagefile, dtype=np.uint8).reshape(
    len(labels), 1, 28, 28)
```

接下来用前面章节介绍过的方法，得到训练数据集的图数据 **images** 和

正确标注数组 **labels**，以及验证用图片 **images_v** 和正确标注 **labels_v**。

4.4.2 卷积神经网络模型的原始实现

超参数定义

在实现模型之前，先定义学习率和数据类型参数，这个数据类型参数决定了模型所有处理层，初始化参数的数值类型，继而影响运算过程中临时变量所占用的内存空间大小：

```
# 基础学习率
LEARNING_BASE_RATE = 0.01
# 设置默认数值类型
DTYPE_DEFAULT = np.float32
```

接下来定义模型各个处理层的超参数：

```
# 输入图片宽和高的尺寸
IMAGE_SIZE = 28
# 输入图片颜色通道数
IMAGE_CHANNEL= 1
# 第一个卷积层的卷积核宽高尺寸
CONV1_F_SIZE = 5
# 第一个卷积层的卷积处理步长
CONV1_STRIDES = 1
# 第一个卷积层的输出尺寸大小不变，仍为28×28
CONV1_O_SIZE = 28
# 第一个卷积层的输出深度，与卷积核个数一致
CONV1_O_DEPTH = 32
# 第一个降采样模板尺寸
POOL1_F_SIZE = 2
# 第一个降采样处理步长
POOL1_STRIDES=2
# 第二个卷积层的卷积核尺寸
CONV2_F_SIZE = 5
# 第二个卷积层的卷积处理步长
CONV2_STRIDES =1
# 第二个卷积层的输出尺寸
```

```
CONV2_O_SIZE = 14
# 第二个卷积层的输出深度增加一倍
CONV2_O_DEPTH = 64
#第二个下采样模板尺寸
POOL2_F_SIZE = 2
#第二个下采样处理步长
POOL2_STRIDES=2
# 第一个全连接层输入维度
FC1_SIZE_INPUT = 3136
# 第二个全连接层输出维度，第三个全连接的输出维度等于
# 分类类别数K，无须设置
FC1_SIZE_OUTPUT = 512
```

超参数定义好之后，就可以基于算法部分介绍过的卷积层与池化层的计算过程，完成卷积层与池化层的原始实现。

实现卷积层处理

卷积层计算过程写成求和形式：

$$z_{d,i,j} = \sum_{\mathrm{ch}=0}^{\mathrm{CH}-1} \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} w_{d,\mathrm{ch},r,c} x_{\mathrm{ch},i+r,j+c} + b_d$$

其中 ch 为输入数据通道（channel），r 和 c 分别为过滤器在高度和宽度上的尺寸。

运算结果经过 ReLU 激活函数，得到的当前卷积层在深度 d 上激活后的输出：

$$a_{d,i,j} = \mathrm{ReLU}(z_{d,i,j})$$

基于上式，容易完成如下原始实现：

```
# 卷积处理的原始实现，输入输出宽高尺寸不变的过滤器
# 当s==1时，p=(f-1)/2
# 入参：
#  x:batch×depth_i×row×col,其中depth_i为输入矩阵深度，
#  depth>1维度=4，depth=1维度=3，对MNIST场景，维度=4
#  w:depth_o×depth_i×row×col，其中row和col为过滤器尺寸，
```

```
#  depth_o为过滤器个数或输出矩阵深度, depth_i和x的depth一致, 可以增加校验
#     b: 长度为depth_o的数组,b的长度即为过滤器个数,
#       和w的depth_o一致, 用于增加校验
#     output_size:卷积输出尺寸
#     strides: 默认为1
# 返回: 卷积层加权输出, 互相关运算(cross-correlation)
#       conv:batch×depth_o×output_size×output_size
def conv(x, w, b, output_size, strides=1):
    batches = x.shape[0]
    input_size = x.shape[2]
    # 过滤器尺寸
    filter_size = w.shape[2]
    # 边缘填充尺寸
    p = ((output_size - 1)*strides+ filter_size-input_size)/2
    p = int(p)
    # 判断是否需要做边缘填充处理
    if p > 0:
        x_pad = padding(x, p)
    else:
        x_pad = x
    conv = np.zeros((batches, w.shape[0], output_size, output_size))

    #逐mini_batch做卷积处理
    for batch in range(batches):
        # 卷积核深度
        for depth_o in range(output_depth):
            # 输出矩阵尺寸
            for i in range(output_size):
                for j in range(output_size):
                    # 使用卷积核逐元素计算点积, 再加上偏置项
                    conv[batch, depth_o, i, j] = (
                    (x_pad[batch, :, i * strides:i * strides + filter_size,
```

```
                    j * strides:j * strides + filter_size]
                        * w[depth_o, :, :, :]
                    ).sum() + b[depth_o]
        )
    # 返回卷积层互相关运算的结果
    return conv
```

对 batches 中的每个 mini-batch, 按照卷积层的不同输出深度，用这个深度上的权值参数矩阵，对每个输入通道上输入矩阵做互相关运算，累加后再和该深度的偏置分量求和。

卷积层的反向传播

```
##bp4conv: conv反向传播梯度计算,直观原始实现
#入参:
#   d_o:卷积输出误差
    batches×depth_o×output_size×output_size,
#     尺寸同conv的输出
#   w:depth_o×depth_i×filter_size×filter_size
#   a:原卷积层输入batch×depth_i×input_size×input_size
#    strides:
#返回:
#  d_i:卷积输入误差batch×depth_i×input_size×input_size,
#  其中 depth_i为输入节点矩阵深度
#  dw:w梯度, 尺寸同w
#  db:b梯度尺寸同b,是depth_0×1数组
def bp4conv(d_o,w,a,strides):
    batches = a.shape[0]
    input_size = a.shape[2]
    depth_i = w.shape[1]
    depth_o = d_o.shape[1]
    output_size = d_o.shape[2]
    f_size = w.shape[2]

    # 翻转卷积核
    w_rtUD = w[:,:,::-1]
    w_rtLR = w_rtUD[:,:,:,::-1]
    w_rt = w_rtLR.transpose(1,0,2,3)
```

```
25      # 误差项传递
26      d_i = conv_efficient(d_o,w_rt,0,input_size,1)
27      #每个d_o上的误差矩阵相加
28      db = np.sum(np.sum(np.sum(d_o, axis=-1),axis=-1), axis
            =0).reshape(-1,1)
29      # dw:
30      #     以d_o作为卷积核，对原卷积层输入a做互相关运算得到
            dw
31      #     d_o的每一层depth_o，作为卷积核 14×14，
32      #    与原卷积输入a的每一个depth_i输入层14×14做互相关运
            算
33      #    得到depth_i个5×5结果矩阵
34      dw = np.zeros_like(w, dtype=DTYPE_DEFAULT)
35      for i in range(depth_o):
36          for j in range(depth_i):
37              for batch in range(batches):
38                  dw[i,j,:,:] += conv_efficient(a[batch,j
                        ,:,:].reshape(1,1,input_size,input_size
                        ),
39                  d_o[batch,i,:,:].reshape(1,1,output_size,
                        output_size),
40                  0,
41                  f_size).reshape(f_size,f_size)
42
43      dw = dw / batches
44      db = db / batches
45      return d_i,dw,db
```

实现池化层处理

池化层的下采样计算：

$$\boldsymbol{z}^{(l)} = \text{subsample}(f(\boldsymbol{z}^{(l-1)}))$$

对 batches 中的每个 mini-batch，按照的不同深度，以池化过滤器尺寸，对每个深度上的输入矩阵做下采样，得到深度不变尺寸缩小的池化层输出。

```
1   # pooling, 池化层用于降采样,直观原始实现
```

```
# 入参:
#   x规格: batch×depth_i×row×col,其中depth_i为输入节点矩阵深度
#   fileter_size: 过滤器尺寸
#   strides: 默认为1
#   type: 降采样类型MAX/MEAN ,默认为MAX
# 返回: 卷积层加权输出(互相关)
#       pooling :batch×depth_i×output_size×output_size
def pool(x, filter_size, strides=2, type='MAX'):
    batches = x.shape[0]
    input_size = x.shape[2]
    output_size = (input_size - filter_size)/strides + 1
    output_size = int(output_size)
    pooling = np.zeros((batches,x.shape[1], output_size, output_size))

    for batch in range(batches):
        for depth_i in range(channels):
            for i in range(output_size):
                for j in range(output_size):
                    # 按照输出尺寸做最大池化层下采样
                    pooling[batch, depth_i, i, j] = (
                    x[batch, depth_i, i * strides:i * strides + filter_size,
                    j * strides:j * strides + filter_size]).max()
    return pooling
```

池化层反向传播

```
# bp4pool: 反向传播上采样梯度
# 入参:
#  dpool: 池化层输出的误差项
#    N×3136 =N×(64×7×7)=batches×(depth_i×pool_o_siz×pool_o_size)
#           reshape为
#    batches×depth_i×pool_o_size×pool_o_size
```

```
6  #  pool_idx : MAX pool时保留的max-value索引
7  #              batches × depth_i × y_o × x_per_filter
8  #  pool_f_size: 池化过滤器尺寸
9  #  pool_stides: 池化步长
10 #  type: MAX/MEAN, 默认为MAX
11 # 返回:
12 #  dpool_i:传递到上一层的误差项,
13 #        batches×depth_i×pool_i_size×pool_i_size
14 #   当strides=2,filter=2时, pool的pool_i_size是pool_o_size
       的2倍
15 def bp4pool(dpool,pool_idx,pool_f_size,pool_strides,type='
       MAX'):
16     batches = dpool.shape[0]
17     depth_i = pool_idx.shape[1]
18     y_per_o = pool_idx.shape[2]
19     x_per_filter = pool_f_size * pool_f_size
20     pool_o_size = int(np.sqrt(y_per_o))
21     input_size = (pool_o_size - 1) * pool_strides +
           pool_f_size
22     dpool_reshape = dpool.reshape(batches, depth_i,
           y_per_o)
23     dpool_i_tmp = np.zeros((batches, depth_i, input_size,
           input_size))
24     pool_idx_reshape = np.zeros(dpool_i_tmp.shape)
25     for j in range(y_per_o):
26         # 对每个输出层的误差做反向传播
27         b = int(j / pool_o_size) * pool_strides
28         c = (j % pool_o_size) * pool_strides
29         #pool_idx_reshape规格同池化层输入
30         #每个数据块的max-value元素位置值为1, 其余值为0
31         pool_idx_reshape[:, :, b:b + pool_f_size, c:c +
               pool_f_size] = pool_idx[:, :, j, 0:x_per_filter
               ].reshape(
32             batches,
33             depth_i,
34             pool_f_size,
35             pool_f_size)
```

```
36          #dpool_i_tmp尺寸同池化层输入，每个数据块的值均以对
                应dpool元素填充
37          # 只需要循环x_per_filter次得到填充扩展后的delta
38          for row in range(pool_f_size):
39              for col in range(pool_f_size):
40                  dpool_i_tmp[:, :, b + row, c + col] =
                        dpool_reshape[:, :, j]
41      # 相乘后，max-value位置delta向上传播，其余位置的delta
            为0
42      dpool_i = dpool_i_tmp * pool_idx_reshape
43
44      # 返回上采样结果，没有参数梯度需要计算
45      return dpool_i
```

边缘填充

在卷积层和池化层处理之前，可能需要对输入张量的每个分量矩阵做边缘填充。

先尝试直接使用 numpy.pad 方法，对输入数据 x 按照给定宽度做边缘填充，默认情况下，填充常量 0。

```
1 # 按照padding尺寸做边缘填充，默认以0填充
2 x_paded = np.pad(x,pad_width,'constant')
```

以上源码是卷积层与池化层的前向计算和反向传播算法的原始实现，逻辑直观明了，有助于理解 CNN 的计算过程。

原始实现，嵌套多重循环，具有多项式时间复杂度，理论上可接受。然而实践上是不可行的，原始算法需要加速改进。在大样本数据集下，只有设法提高运算效率，才能方便地调参和迭代训练。

4.5 小结

这一章描述了卷积神经网络的结构与核心算法，首先介绍了 CNN 中的卷积与数学上经典卷积定义和算法的不同之处；然后介绍了卷积神经网络输入数据的规格，并对 MNIST 数据的预处理做了相应调整；最后讨论了卷积层、池化层的一种原始实现。

基于这些算法，虽然可以从底层完整实现一个多层卷积神经网络，然而从实践考量将迎来两个新的挑战。

首先，CNN 较之前的多层神经网络，虽然通过权值共享和降采样处理，压缩了参数量和运算量，可是需要通过增加过滤器的深度来抽取更丰富的高层图像特征，新引入卷积和池化运算，使训练成本高企。

其次，输入数据可能包含多个通道，使训练的运算中间变量成倍增加；在反向传播过程中，除了需要缓存各层级训练得到的权值参数，还需要缓存反向传播的误差张量和最大池化层的 max-value 索引矩阵，内存开销制约 mini-batch 的容量上限，训练模型参数时，各方向上的更新梯度容易相互抵消，模型收敛缓慢。

下一章将描述为卷积神经网络核心算法提速的方法，通过改进池化层与卷积层的原始算法，使前向和反向传播大幅提速。再为批次梯度下降优化算法，引入动量因子和自适应方法，提高优化的效率，使模型快速收敛，将 MNIST 数据集上的识别正确率进一步提高到 99% 以上。

拓展阅读

斯坦福大学的深度学习教程，介绍了非监督特征学习和深度学习的基本概念和算法。其中的卷积神经网络部分，对卷积层和池化层处理有详细的介绍。
The Stanford University, Unsupervised Feature Learning and Deep Learning tutorial.
`http://ufldl.stanford.edu/tutorial`

马里兰大学的图像处理课程（CMSC 426）, 第三单元：过滤和互相关处理，讲解了经典卷积和互相关的区别。
CMSC 426 Image Processing （Computer Vision）,Class 3：Filtering and Correlation.
`http://www.cs.umd.edu/~djacobs/CMSC426/CMSC426_10.htm`

参 考 文 献

[1] Patrice Simard，David Steinkraus，John C Platt. **Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis**. Proceedings of the International Conference on Document Analysis and Recognition.2003:958-962.

[2] Krizhevsky Alex , Sutskever I, Geoffrey Hinton. **ImageNet Classification with Deep Convolutional Neural Networks**. NIPS. Curran Associates Inc. 2012.

[3] Yann LeCun, Yoshua Bengio, Geoffrey Hinton. **Deep learning**. Nature, 2015，521(7553):436–444.

[4] Murugan, Pushparaja. **Feed Forward and Backward Run in Deep Convolution Neural Network**. arXiv:1711.03278v1. 2017.

第 5 章

卷积神经网络
——算法提速和优化

“我欲因之梦吴越，一夜飞渡镜湖月。”

——[唐] 李白《梦游天姥吟留别》

上一章为模型引入卷积处理，介绍了卷积层、池化层的前向传播，以及反向传播算法；这一章，仍然**不借助深度学习框架**，先改进算法的原始实现，采用多维数组切片和**向量化**方法加速卷积层和池化层的处理，再描述**学习策略的优化改进方法**，显著提高收敛效率，把 MNIST 数据集上的预测正确率进一步提高到 99% 以上。

5.1 第一个挑战：卷积神经网络的运算效率

基于上一章介绍的算法，虽然可以从底层完整地实现一个多层卷积神经网络。然而，从实践考量又有新的挑战：

首先，CNN 较之前的多层神经网络，虽然通过权值共享和降采样处理大幅减少了参数量，降低了运算消耗，可是仍然需要通过增加过滤器深度，即卷积核数量，来抽取更丰富的高层图像特征；卷积和池化处理的原始实现，嵌套了多重循环，具有多项式时间复杂度，使训练成本高企，理论上可接受，然而实践不可行。

在大容量数据集下，只有提高训练效率，才能方便地调整模型参数和迭代训练，原始算法的实现需要提速改进。工程实践中，要想提高模型在大容量数据集上的训练效率，有两种直接且有效的方法——提升硬件算力和采取架构的并行化设计。这两种方法都有一个共同的前提，即支撑神经网络模型的基础算法本身应该是合理且高效的。这一章再次深入到卷积层、池化层处理的核心算法，探讨优化的思路，落地为具体措施，再对所讨论的每一类优化措施均给出一种**不借助深度学习框架**的实现，以供参考。

5.2 提速改进

通过特定编译器和算法改进都可以起到提速的效果。

Numba 库

一种简便的加速方法是用 Numba 的 **JIT** 编译器。把 Python 和 NumPy 代码转换为效率更高的本地机器码，以加快执行速度。这种方法对含有大量数组、矩阵的循环数值运算，效果显著。

Numba 支持装饰器语法，很好地封装了优化细节，对已实现的算法逻辑是透明的，易于使用，本书不再展开。

接下来讨论的加速方法，多从算法层面着手，不依赖本地硬件，效率优于在原始算法直接采用 JIT 编译的结果，改进算法实现后，还可以进一步采用 JIT 或支持 CUDA 的 NumPy 衍生库，做叠加提速。

在优化之前，先观察算法各个环节的运算消耗对比情况，参见表 5.1。

表 5.1 包含两个 conv-pool 单元的卷积神经网络，原始实现的运算消耗对比，卷积和池化运算复杂，耗时较多，全连接和 Softmax 运算较快

序号	算法环节	超参数定义	时间片占比
1	第一个卷积层	32 个 5×5 卷积核，边缘填充	53.22%
2	第二个卷积层	64 个 5×5 卷积核, 边缘填充	34.85%
3	第一个下采样层（最大池化）	2×2 采样	7.95%
4	第二个下采样层（最大池化）	2×2 采样	3.83%
5	第一个全连接层	权值参数 3136×512	0.01%
6	第二个全连接层	权值参数 512×10	较小，可忽略不计
7	Softmax 处理	无参数	较小，可忽略不计

从表5.1可以看到，超过 88% 的运算量被消耗在卷积处理环节，其次是最大池化层的下采样处理，接近总运算量的 12%，相比之下，全连接层、激活函数、Softmax 和各个处理层之间的尺寸转换（reshape）变换处理，只占用了总运算时间的万分之一。

图5.1更加直观地对比了这个模型中各个环节的运算时间消耗。

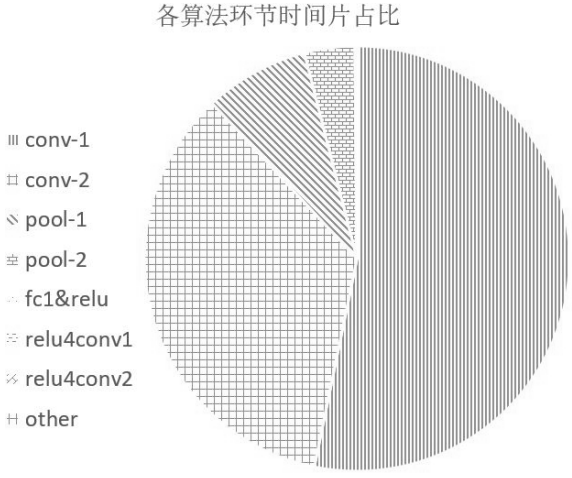

图 5.1 卷积神经网络，原始实现，大部分运算量在做卷积和池化处理环节

这个运算时间的占比，仅是对前向传播各个算法环节的分析。回顾卷积层的反向传播运算，本质上仍然是翻转卷积核之后的互相关运算。可想而知，算

法提速的关键，是卷积运算和池化运算环节，而在这两种运算中，每个深度都可能会调用到的边缘填充处理，则是首当其冲需要考虑提速的运算步骤。

不妨先从卷积和池化的公共底层方法——边缘填充开始。

5.2.1 边缘填充提速

卷积运算和池化运算之前，都有可能需要对输入数据做边缘填充，而且边缘填充要在每个输出深度上进行，表5.1所示的例子中，池化层恰好不需要做边缘填充，但是两个卷积层处理仍然要做 32+64=96 个输出深度上的边缘填充，这 96 个深度上的边缘填充是每个 mini-batch 中全部样本都要执行的运算步骤。

原始实现中，我们直接调用了 numpy.pad() 方法实现边缘填充：

```
# 使用NumPy的内置方法做zero-padding
x_paded = np.pad(x,pad_width,'constant')
```

NumPy 是一套成熟的科学计算库，已经做了大量性能优化，然而自带的 pad() 方法是为了满足通用处理而设计的，观察方法复杂的入参定义，你可以设想：这个通用方法需要适配各种各样的矩阵填充场景，那么对每一类特定的场景，必然有大量的冗余判断逻辑，因此如果针对 CNN 模型所需要的场景**定制实现**边缘填充逻辑，应该是**有提速空间**的。

实践出真知。

```
def padding(x, pad):
    size_x = x.shape[2] #输入矩阵尺寸
    size = size_x + pad*2 # 边缘填充后尺寸
    #x的0维是mini-batch
    if x.ndim ==4: # 每个元素是3维的，有颜色通道
        # 初始化同维全0矩阵
        padding = np.zeros((x.shape[0],x.shape[1],size,
            size))
        # 中间以x填充
        padding[:,:, pad: pad + size_x, pad: pad + size_x]
             = x

```

```
11      elif x.ndim == 3 : # 每个元素是2维的，只有长宽，没有颜
            色通道
12          padding = np.zeros((x.shape[0],size,size))
13          padding[:, pad: pad + size_x, pad: pad + size_x] =
                x
14
15      return padding
```

对比在 CNN 场景下，重新定制实现的边缘填充方法的输出，与 NumPy 自带方法的输出数值结果完全一致。再对比两个边缘填充方法的执行效率，果然有 **20 倍速度的提升**。

```
1  %timeit x_pad = padding(x,1)
2  %timeit x_pad1 = np.pad(x,((0,0),(0,0),(1,1),(1,1)),'
       constant')
3  np.allclose(padding(x,1),np.pad(x,((0,0),(0,0),(1,1),(1,1)
       ),'constant') )
4  # 效率对比
5  12.2 µs ± 1.2 µs per loop (mean ± std. dev. of 7 runs,
       100000 loops each)
6  243 µs ± 13.9 µs per loop (mean ± std. dev. of 7 runs,
       1000 loops each)
7  # 填充一致性比较结果
8  True
```

接下来为池化层提速。

5.2.2 池化层提速

池化层提速的思路可放在多重循环上。池化层的下采样计算：

$$z^{(l)} = \text{subsample}(f(z^{(l-1)}))$$

如果采用 NumPy 的 ndarray 核心对象处理多维数组，用切片方法替换逐元素的循环语法，可以将算法的多项式时间复杂度从高阶降为低阶。

```
1  # 优化说明：
2  #  1.先把x组织成特定维度：
```

```
3  #   batch×depth_i×(output_size×output_size)×
4  #   (filter_size×filter_size)
5  #  2.然后利用矩阵运算，对最后一个维度获取最大值，得到
6  #   batch×depth_i×(output_size×output_size)
7  #  3.再转换维度为 batch×depth_i×output_size×output_size
8  #  4.保存每次max value的index用于反向传播
9  # 入参:
10 #  x: batch×depth_i×row×col,其中depth_i为输入矩阵深度
11 #  fileter_size:过滤器尺寸
12 #  strides: 步长，设置默认为1
13 #  type: 降采样类型，MAX/MEAN，默认为MAX
14 # 返回: 下采样输出
15 #   pooling : batch×depth_i×output_size×output_size
16 #   pooling_idx : batch×depth_i×y_per_o_layer×x_per_filter
17 #     其中 y_per_o_layer=output_size×output_size
18 #     x_per_filter=pool_f_size×pool_f_size
19 #     在当前input_block对应max-value位置为1,其他为0
20 def pool(x, filter_size, strides=2, type='MAX'):
21     batches = x.shape[0]
22     depth_i = x.shape[1]
23     input_size = x.shape[2]  #
24     x_per_filter = filter_size * filter_size
25     output_size = int((input_size - filter_size) / strides
           ) + 1
26     #输出矩阵每一层元素个数
27     y_per_o_layer = output_size * output_size
28     x_vec = np.zeros((batches, depth_i, y_per_o_layer,
           x_per_filter))
29     #池化处理
30     for j in range(y_per_o_layer):
31         # MNIST strides ==2 按输入深度优化
32         b = int(j / output_size) * strides
33         c = (j % output_size) * strides
34         x_vec[:, :, j, 0:x_per_filter]=x[:,:,b:b+strides,c
               :c+strides].reshape(batches,depth_i,
               x_per_filter)
35
```

```
36      pooling = np.max(x_vec,axis=3).reshape(batches,
            depth_i, output_size, output_size)
37      # 保留采样索引
38      pooling_idx=np.eye(x_vec.shape[3])[x_vec.argmax(3)]
39      # 返回下采样结果和采样索引
40      return pooling,pooling_idx
```

池化层实现需要注意保留 max-value 位置索引，在误差反向传播的时候，可以复用在池化层的上采样运算环节。

比较原始池化运算和改进后的算法, 得到**近 3 倍的速度提升**。

```
1  1.01 ms ± 122 µs per loop (mean ± std. dev. of 7 runs,
       1000 loops each)
2  375 µs ± 56.3 µs per loop (mean ± std. dev. of 7 runs,
       1000 loops each)
```

完成了边缘填充和池化层处理的算法提速，再来看耗时占比最高的卷积层处理环节。

5.2.3 卷积层处理

卷积神经网络中，卷积层处理的本质是互相关运算，而互相关运算在前向计算和反向传播中都要用到，模型的运算量主要集中在这个基础算法。由此可见，卷积层的互相关运算是提升模型训练效率的关键节点。

首先，观察卷积层互相关运算，计算过程的求和形式：

$$z_{d,i,j} = \sum_{\mathrm{ch}=0}^{\mathrm{CH}-1} \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} w_{d,\mathrm{ch},r,c} x_{\mathrm{ch},i+r,j+c} + b_d$$

其中 ch 表示输入数据的通道（channel），r 和 c 分别是卷积过滤器（卷积核）高度和宽度的尺寸。

卷积层运算结果经过 ReLU 激活函数，得到的当前卷积层在深度 d 上激活后的输出：

$$a_{d,i,j} = \mathrm{ReLU}(z_{d,i,j})$$

可以使用与池化层优化一样的思路，采用 NumPy 的 ndarray 核心对象切片方

法，减少多重循环层次。此外，卷积层处理还有一种有效的提速方法：先对输入数据和权值参数做向量化处理，把输入数据每一个需要做卷积的数据块展开成列向量；再把权值参数张量的每个分量矩阵展开成行向量；经过向量化预处理，卷积运算就成了两个大矩阵的乘积运算。

例如：

1. 宽高均为 227 像素的三通道（channel=3）彩色图片，与每个 $11\times11\times3$ 卷积核做步长为 4 的卷积运算，则输出结果矩阵的宽高均为 $(\text{size}_i - \text{size}_{\text{filter}} - 2P)/\text{strides}+1 = (227-11-0)/4+1 = 55$，而输出矩阵展开后尺寸为 $55\times55 = 3025$；对输入数据，每个数据块的像素数为 $11\times11\times3 = 363$，输入图像用于做卷积运算的数据块可以展开为 363×3025 的输入矩阵。

2. 对于 96 卷积核的过滤器，权参张量可以展开成尺寸为 96×363 的权值参数矩阵。

3. 两个矩阵乘积运算，得到 96×3025 的结果矩阵，还原为 $96\times55\times55$，激活之后，就是这张图片的卷积层处理结果。

实际训练过程中，输入数据往往不是单张图片，而是 mini-batch 集合，需要先为表达输入样本的多维数组切片增加 mini-batch 维度，再做向量化处理：

```
# 说明：对卷积层的原始输入做向量化处理
#入参
#    x:padding后的实例
    batches×channel×conv_i_size×conv_i_size
#    fileter_size : 过滤器尺寸
#    conv_o_size: 卷积层输出尺寸
#    strides:   卷积层步长
# 返回
#    x_col: batches×(channel×filter_size×filter_size)×
#   ( conv_o_size×conv_o_size)
# 优化措施
# 1.每个底层循环，更新depth_i行
# 2.每一行，一次更新filter_size_列
# 3.每次更新一个filter_size block
```

```
14 # 4.不再做输入通道维度上的循环，直接转换维度(reshape)并赋值
15 def vectorize4conv_batches(self, x, filter_size, conv_o_size, strides):
16     batches = x.shape[0]
17     channels = x.shape[1]
18     x_per_filter = filter_size * filter_size
19     # 计算列向量展开后的长度
20     shape_t = channels * x_per_filter
21     # 初始化展开后的列向量
22     x_col = np.zeros((batches, channels * x_per_filter, conv_o_size * conv_o_size), dtype=self.dataType)
23     for j in range(x_col.shape[2]):
24         b = int(j / conv_o_size) * strides
25         c = (j % conv_o_size) * strides
26         # 使用ndarray核心对象的切片方法替换多重循环
27         # 把输入数据块展开为列向量
28         x_col[:, :, j] = x[:, :, b:b + filter_size, c:c + filter_size].reshape(batches, shape_t)
29 
30     # 返回向量化结果
31     return x_col
```

卷积核上的权参张量和输入数据的 mini-batch 容量无关，可以直接把权参 $\boldsymbol{W}$ 在每个深度上的分量，展开成行向量：

```
1 w_row = w.reshape(depth_o, x_col.shape[1])
```

接下来，就可以直接计算两个大矩阵的乘积，实现互相运算，完成卷积层的前向计算处理：

```
1 # 初始化互相关运算结果
2 conv = np.zeros((batches, depth_o, (output_size * output_size)), dtype=self.dataType)
3 # 此处不在batch维度使用广播机制，效率要好于切片方法
4 for batch in range(batches):
5     conv[batch] = Tools.matmul(w_row, x_col[batch]) + b
```

对比卷积处理的原始算法和提速后的算法，在数值结果一致的前提下，得到 **4 倍提速**，进一步使整个模型的运算耗时减少到提速前的三分之一。

```
758 µs ± 62.1 µs per loop (mean ± std. dev. of 7 runs,
    1000 loops each)
182 µs ± 13.3 µs per loop (mean ± std. dev. of 7 runs,
    10000 loops each)
```

由于卷积运算被转换为大矩阵的乘积运算，这个算法还可以采用 BLAS 基础线性代数程序集的硬件加速版本，利用存储器层面的优化进一步提速。

以上是池化和卷积处理的原始算法经过提速优化后的改进实现。

接下来看反向传播算法的具体实现。

5.3 反向传播算法实现

最后两个全连接层反向传播算法，复用第 3 章已实现的方法，此处不再累述；这里，把反向传播优化的重点，放在运算消耗较大的卷积层和池化层反向传播环节。先看处理较为简单的池化层反向传播上采样运算。

5.3.1 池化层反向传播

池化层的上采样处理:

$$\boldsymbol{\delta}^{(l-1)} = \text{upsample}(\boldsymbol{\delta}^{(l)} \otimes f^{'}(\boldsymbol{z}^{(l-1)})$$

平均池化层反向传播相对简单，只要使用误差矩阵对应位置元素，对继续回传的误差矩阵直接做上采样填充即可。

最大池化层反向传播处理稍复杂一些，需要用到池化层前向处理时预留的 max-value 位置索引。计算误差矩阵梯度的时候，在各层误差矩阵中处于 max-value 位置索引上的元素, 为其填充回传的误差值，而对处于位置索引之外的元素，因为前向传播时没有参与计算，不影响反向传播的梯度，直接用 0 填充，然后继续向上传递：

```
for j in range(y_per_o):
    b = int(j / pool_o_size) * pool_strides
    c = (j % pool_o_size) * pool_strides
```

```
    # pool_idx_reshape规格同池化层输入,
    # 每个数据块的max-value位置值为1, 其余位置值为0
    pool_idx_reshape[:, :, b:b + pool_f_size, c:c +
        pool_f_size] = pool_idx[:, :, j, 0:x_per_filter].
        reshape(
    batches,
    depth_i,
    pool_f_size,
    pool_f_size)
# dpool_i_tmp规格规格同池化层输入
# 每个数据块的值均以对应dpool元素填充
# 只需要循环 x_per_filter次,即可得到填充扩展后的delta
    for row in range(pool_f_size):
        for col in range(pool_f_size):
        dpool_i_tmp[:, :, b + row, c + col] =
            dpool_reshape[:, :, j]
# 相乘后, max value位置delta向上传播, 其余位置delta为0
dpool_i = dpool_i_tmp * pool_idx_reshape
```

5.3.2 卷积层反向传播

卷积层反向传播需要在翻转卷积核后，用新的卷积核同误差矩阵做互相关运算，仍然可以复用向量化方法提速后的互相关运算来完成核心运算步骤。

卷积层，误差矩阵向上传递:

$$\begin{aligned}\delta_{i,j}^{(l-1)} &= \frac{\partial \text{Loss}^{(l)}}{\partial z_{i,j}^{(l-1)}} \\ &= \sum_{r=0}^{R-1}\sum_{c=0}^{C-1} w_{r,c}^{(l)}\delta_{i-r,j-c}^{(l)} f'(z_{i,j}^{(l-1)})\end{aligned}$$

需要先对卷积核做 180° 翻转:

$$\frac{\partial \text{Loss}^{(l)}}{\partial a_{i,j}^{(l-1)}} = \boldsymbol{\delta}^{(l)} \odot \text{rot180°}(\boldsymbol{W}^{(l)})$$

翻转卷积核可以采取如下简洁的步骤：先把卷积核做上下和左右两次翻

转，再对前两个维度做高维转置，重新排列分量矩阵的参数值，即可实现 180° 翻转，得到新的卷积核。使用 NumPy 中 ndarray 核心对象的切片方法，可以快速完成卷积核的两次翻转运算。

```
# 上下翻转
w_rtUD = w[:,:,::-1]
# 左右翻转
w_rtLR = w_rtUD[:,:,:,::-1]
# 0维和1维互换实现高维转置
w_rt = w_rtLR.transpose(1,0,2,3)
```

接下来就可以复用提速改进后的互相关（cross-correlation）运算实现卷积层的误差矩阵向上传递。

计算卷积核权参更新梯度 $\nabla \boldsymbol{W}$

以卷积层误差张量 $\boldsymbol{\delta}$ 作为过滤器，和上一层激活后输出做互相关运算：

$$\frac{\partial \text{Loss}^{(l)}}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial \text{Loss}^{(l)}}{\partial \boldsymbol{z}^{(l)}} \frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{a}^{(l-1)} \dot{\boldsymbol{\delta}}^{(l)}$$

展开为逐项累加形式：

$$\frac{\partial \text{Loss}^{(l)}}{\partial w_{i,j}^{(l)}} = \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} \delta_{r,c}^{(l)} a_{i+r,j+c}^{(l-1)}$$

复用向量化处理, 再用误差矩阵对原始输入做互相关运算, 得到 $\nabla \boldsymbol{W}$ ：

```
 # 卷积核展开为行向量
w_row = w.reshape(batches, depth_o, x_per_filter)
# 初始化误差张量
conv = np.zeros((batches, depth_i, depth_o, (output_size *
     output_size)), dtype=self.dataType)
# 用大矩阵的乘积代替反向传播的卷积运算
for batch in range(batches):
    for col in range(depth_i):
        conv[batch,col] = Tools.matmul(w_row[batch],x_col[
            batch, col])
conv_sum = np.sum(conv, axis=0)
```

```
#先转置（transpose）再转换尺寸（reshape），避免卷积层反向传播错位
conv = conv_sum.transpose(1, 0, 2).reshape(depth_o, depth_i, output_size, output_size)
```

计算卷积核的偏置梯度 $\nabla \boldsymbol{b}$

$$\frac{\partial \text{Loss}^{(l)}}{\partial \boldsymbol{b}_d^{(l)}} = \frac{\partial \text{Loss}^{(l)}}{\partial \boldsymbol{z}_d^{(l)}} \frac{\partial \boldsymbol{z}_d^{(l)}}{\partial \boldsymbol{b}_d^{(l)}} = \sum_i \sum_j \delta_{d,i,j}^{(l)}$$

有 D 个过滤器的卷积核，在顺位 d 这个卷积核对应偏置项的梯度，是该层误差张量 $\boldsymbol{\delta}$ 在顺位 d 这个分量矩阵上各个元素的和：

```
# 每个d_o上的误差矩阵相加
db = np.sum(np.sum(np.sum(d_o, axis=-1), axis=-1), axis=0).reshape(-1, 1)
```

以上是 CNN 反向传播处理中主干算法的实现。提速改进之后算法的运算效率比原始算法有了大幅提高。

主流深度学习框架随着版本更新，所提供的接口改版层出不穷，然而这些参数各异的接口，都离不开底层基础算法的支撑。这一章对基础算法所做的提速改进，仅仅是诸多方法中的一种，尝试对核心算法做分析和改进，有助于理解算法的本质，进而在工程实践中更好地运用深度学习框架。

5.4 第二个挑战：梯度下降的幅度和方向

之前的模型采用随机梯度下降学习策略，可以得到不错的收敛效果；引入卷积处理后，输入包含多个通道，输出也随多个卷积核而增加了深度，使参数量和运算量都大幅上升，模型训练时中间结果占用的内存也相应增加了；在反向传播过程中，除了需要缓存各层级的权值参数和误差张量，对最大池化层处理还需保持 max-value 位置索引矩阵，导致内存用量增加。这些因素使每次训练样本的 mini-batch 容量受到限制而不能过大，若采用小批量样本训练模型参数又使得模型收敛缓慢，在梯度下降的幅度和方向上带来挑战。

1) 首先，如果梯度更新的幅度过小，参数更新和模型收敛都变得缓慢；而梯度更新的幅度过大，又可能使模型参数在优化的目标区域附近反复移动，难

以收敛。

2) 再者，使用批次梯度下降，小容量 mini-batch 得到的参数更新梯度，更有可能在不同的梯度方向上往复波动，CNN 的层次（深度）越深，这个问题越突出。

第一个问题，可以用递减学习率来缓解。

5.5 递减学习率参数

通过设置逐轮减小的学习率可以控制参数更新幅度，使得训练前期快速接近参数极优值，后期又不会在极优值附近大幅波动。

Yann LeCun 和同事设计的 LeNet-5 模型，开创了卷积神经网络大规模落地商用的成功先河。这个模型在 MNIST 数据集上使用了表 5.2 所示的逐轮次递减学习率来训练模型参数。

表 5.2 LeNet-5 模型的递减学习率参数，
前期快速接近极优值，后期尽量避免大幅波动

训练轮数	递减学习率
0-1	0.005
2-4	0.0002
5-7	0.0001
8-11	0.000 05
12-thereafter	0.000 01

这一系列学习率被全局应用在 LeNet-5 模型中的所有的各层权值参数和偏置项的迭代更新上。

对第二个问题，即在深度模型上使用小容量 mini-batch，得到的参数更新梯度在不同的梯度方向上往复波动的情况，有几种效果较好的学习策略改进方法可以改善。

5.6 学习策略的优化方法

深度学习的学习策略，在随机梯度下降方法的基础上，还有一些优化方法值得尝试。这些方法在适用场景下可以收到种立竿见影的效果，其实现又非常

简洁，下面尽可能按照优化方法之间的逻辑关系依次讨论和实现。

5.6.1 动量方法

动量方法为梯度更新引入过去时刻的速度来缓解 mini-batch 更新梯度在不同方向上往复波动的问题。

这个方法的原理，可以用小球从山上向谷底滚动的例子来类比。假设小球的运动既有向下的加速度，又存在风阻（γ）的反向作用, 在下降的过程中，达到均衡终速（terminal velocity）之前，小球下降的速度是越来越快的。

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \tag{5.1}$$

$$\theta = \theta - v_t \tag{5.2}$$

同样，各个 mini-batch 的参数梯度，在同方向上加速更新，不同方向上减速更新，从而加快收敛，抑制波动。

Momentum 动量方法的实现比较简单：

```
1 v = gamma * v + lr * dx
2 x += - v
```

超参数γ（通常设为 0.9）在优化策略的论述中，常被称为**动量**（Momentum）。实际上，γ 更接近物理意义上的**阻力**，起到的作用是制动和减少动能（kinetic energy），使得小球最终停在谷底。

5.6.2 NAG 方法

NAG 方法在动量方法的基础上更进了一步。

设想沿着山势下降的小球，如果能预判下一时刻的位置，它会在坡度即将变缓的时候减慢速度。由于当前时刻的动量项（momentum term）是已知的，用当前参数减去已知的动量项，可以近似得到下一个位置。这样就能用预判所得近似新位置来计算当前参数的更新梯度。

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \tag{5.3}$$

$$\theta = \theta - v_t \tag{5.4}$$

工程实践中，NAG 有另一个变体：保留动量方法更新速度的方式，仅用预判得到的近似新位置更新参数。

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \tag{5.5}$$

$$\theta = \theta + \gamma v_{t-1} - (1+\gamma) v_t \tag{5.6}$$

NAG 方法的实现也不复杂：

```
vt = gamma * v + lr * dx
x += gamma* v - (1.+gamma) * vt
```

5.6.3 Adagrad 方法

Adagrad 方法是一种自适应学习率的优化方法，在迭代的不同轮次，学习率乘以不同的递减系数。

对于第 t 轮迭代中的第 i 个参数, 其更新梯度 g 为

$$g_{t,i} = \nabla \theta_t J(\theta_{t,i}) \tag{5.7}$$

对比 SGD 算法的参数更新方式：

$$\theta_{t+1,i} = \theta_{t,i} - \eta \; g_{t,i} \tag{5.8}$$

Adagrad 改进为，用历史梯度的均方根 (**Root Mean Squared**) 作为学习率的递减系数：

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\sum_{t=1}^{T} g_{t,i}^2}} \otimes g_{t,i} \tag{5.9}$$

其中“$\otimes$”是表示 Hadamard 乘积运算，计算方法是矩阵对应位置逐元素两两相乘（element-wise product between matrices）。

在初始梯度极小的情况下，为了避免出现除 0 错, 再为分母增加平滑项 ϵ

（smoothing term），因此，表达式被进一步写为

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\sum_{t=1}^{T} g_{i,t}^2 + \epsilon}} \otimes g_{i,t} \tag{5.10}$$

Adagrad 方法的实现：

```
g += dx ** 2
# 用历史梯度的均方根（Root Mean Squared）作为学习率的递减系数
x += - lr * dx / (np.sqrt(g) + eps)
```

Adagrad 的缺点是：分母在模型训练过程中不断增加，导致参数更新的梯度，在后期可能变得非常小，直到模型无法再从误差损失中获取参数的更新量，为了解决这个问题，Geoff Hinton 提出了 RMSprop 方法。

5.6.4 RMSprop 方法

RMSprop 是 Geoff Hinton 在公开在线课程 Coursera（https://www.coursera.org）项目的讲义上提出的方法，这种方法通过为分母中均方根的计算增加 Gamma 系数，来解决 Adagrad 方法中参数更新梯度趋小的问题。

$$E[g_{t,i}] = \gamma E[g_{t-1,i}] + (1-\gamma) g_{t,i}^2 \tag{5.11}$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[g_{t,i}] + \epsilon}} \cdot g_{t,i} \tag{5.12}$$

RMSprop 方法的实现：

```
eg =gamma * eg + (1-gamma) * (dx** 2)
x += - lr * dx / (np.sqrt(eg) + eps)
```

在讲义中，Geoff Hinton 建议超参数 $\gamma = 0.9$。

5.6.5 AdaDelta 方法

AdaDelta 方法另辟蹊径，解决 Adagrad 方法后期参数更新梯度趋小的问题，同时引入自适应机制，从而不再需要设置全局学习率超参数。

$$E[g_{t,i}^2] = \rho E[g_{t-1,i}^2] + (1-\rho) g_{t,i}^2 \tag{5.13}$$

回顾 SGD 的参数更新梯度：

$$\nabla\theta_{t,i} = -\eta \cdot g_{t,i}$$
$$\theta_{t+1,i} = \theta_{t,i} + \nabla\theta_{t,i}$$

Adagrad 方法给学习率 η 增加了历史梯度的均方根 (RMS) 衰减系数：

$$\nabla\theta_{t,i} = -\frac{\eta}{\sqrt{\sum_{t=1}^{T} g_{i,t}^2 + \epsilon}} \cdot g_{t,i} \tag{5.14}$$

而 AdagDelta 不使用梯度平方，改用参数的平方做滑动平均：

$$E[\nabla\theta_i^2]_t = \rho E[\nabla\theta_i^2]_{t-1} + (1-\rho)\nabla\theta_{t,i}^2 \tag{5.15}$$

参数的均方根误差（RMS error）成为

$$\text{RMS}[\nabla\theta]_t = \sqrt{E[\nabla\theta_i^2]_t + \epsilon} \tag{5.16}$$

当前 $\text{RMS}[\nabla\theta_i]_t$ 未知，可以用上一步更新的参数来近似，得到计算参数梯度的新系数，以替换全局学习率 η：

$$\nabla\theta_{i,t} = -\frac{\text{RMS}[\nabla\theta]_{t-1}}{\text{RMS}[g]_t} \cdot g_{t,i} \tag{5.17}$$

AdaDelta 方法的实现：

```
eg =rho * eg + (1-rho) * (dx** 2)
etsq=rho * etsq + (1-rho) * (dt** 2)
dt = np.sqrt( (etsq + eps)/(eg +eps)) * dx
```

```
x += - dt
```

通常设置衰减率超参数 $\rho = 0.9$, 首轮迭代时用 0 初始化计算依赖变量。

上述方法在实践中各有特点，在给参数做梯度下降求解的过程中，空间中在一个维度上有对称的正向斜率，另一个维度上有负向斜率的**鞍点**（saddle point）位置，SGD 方法容易被困住，动量方法和 NAG 方法能够缓慢摆脱出来，而自适应学习率类优化方法，如 Adgrad、RMSprop 及 AdaDelta 方法，可以快速摆脱鞍点，继续沿负梯度方向下降。

5.6.6 Adam 方法

Adam 方法兼用了动量方法的均值（first moment）和自适应方法的均方根（second moment）做参数估计：

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{5.18}$$

$$v_t = \beta_2 m_{t-1} + (1 - \beta_2) g_t^2 \tag{5.19}$$

上式中，由于 m 和 v 预先初始化为全 0 向量，会造成初始几个迭代轮次，学习率的递减系数较小的时候，每次用来更新梯度的偏移量过小，甚至会趋近于 0。针对这一观察，Adam 方法的作者对均值和均方根系数引入了偏差矫正机制（bias correction mechanism）：

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{5.20}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{5.21}$$

之后，再进行逐轮次参数更新：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t \tag{5.22}$$

Adam 方法的实现：

```
m = beta1 * m + (1 - beta1) * dx
mt = m / (1 - beta1 ** t)
v = beta2 * v + (1 - beta2) * (dx ** 2)
```

```
vt = v / (1 - beta2 ** t)
x += - lr * mt / (np.sqrt(vt) + eps)
```

Adam 方法的超参数，通常的建议取值是 $\beta 1 = 0.9$, $\beta 1 = 0.999$。

5.6.7 各种优化方法的比较

实践中，可以尝试各种不同的优化方法，Adam 方法被普遍认为是效果较好的首选方法，然而这个观点也不乏争议。

为了观察各种方法在同一套数据集上的表现，构建一个简单的两层全连接神经网络，第一层包含 512 个隐藏节点，第二层输出和分类类别对应，使用交叉熵损失函数对 MNIST 数据做分类识别。在模型中，为每一种优化方法都设置一套独立的训练参数，采用相同的方法做参数初始化。

训练过程中，每个 epoch 内，训练数据仍然打乱并随机分组，设置 mini-batch 容量为 256，确保对于每个训练迭代步骤，模型都送一组相同的 mini-batch 给各个优化方法，观察这些方法在各自一套独立参数上的优化表现。

图5.2看上去，在合适的学习率选择下，Adam 方法收敛得更快, 模型正确率也更高。

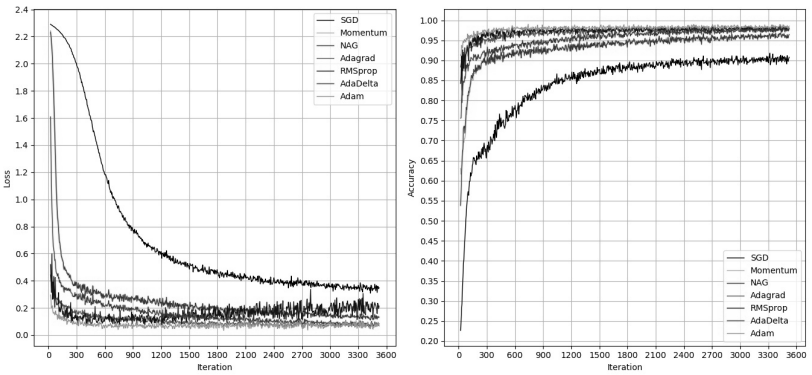

图 5.2　SGD 方法，和其他六种改进优化方法比较。
左图为独立测试集上的损失曲线，右图为推理预测正确率曲线

然而拿到一组数据，往往不知道到底设置多大的初始学习率比较合适，需要尝试不同的学习率。在反复训练代价比较高的情况下，如何开始做模型的训练和调优呢？

仍然用上面这个场景来做分析，这一次，我们观察各个优化方法在不同学

习率下的收敛情况，如图 5.3所示。

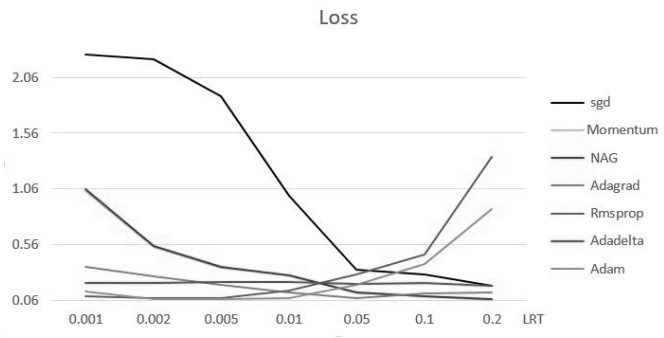

图 5.3 优化方法比较：以当前任务场景在不同的初始学习率下，Loss 比较在较大学习率下，Adam 方法表现不如其他优化方法

在这个任务场景下，如果选择不同的初始学习率，Adam 方法和其他优化方法相比，**不见得更好，反而可能是较差的**，正确率比较参见图5.4。

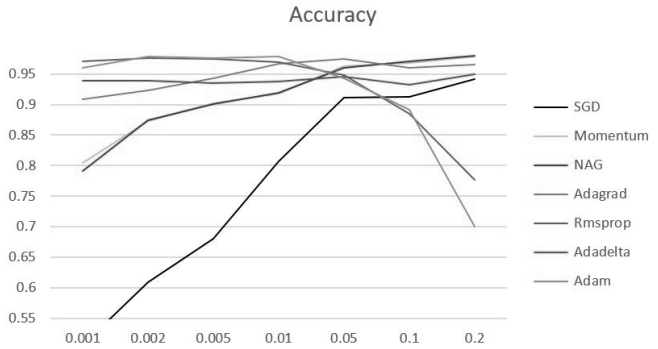

图 5.4 优化方法比较：以当前任务场景在不同的初始学习率下,Acc 曲线比较，Adam 方法不见得更好，AdaDelta 方法的表现最稳定

如果初始学习率设置偏小，SGD 方法的收敛速度明显较慢，Adam 和 RMSprop 方法收敛速度就快多了；然而，逐步调大初始学习率，可以观察到动量方法和 NAG 方法的收敛效果和其他方法相比较逐渐好转；而 Adam 和 RMSprop 方法，反而收敛较慢，甚至不如 SDG 方法的表现。

Adam 和 RMSprop 方法调整了均方根（second moment）计算的系数，避免了参数更新梯度趋小的问题，也使得方法本身对学习率参数更敏感，影响了大学习率下的收敛效果。

而 Adadelta 方法，由于不依赖学习率参数，所以用它做比较基线，容易观察到：较小的初始学习率下，RMSprop 和 Adam 方法表现较好；而较大的

初始学习率下，NAG 和动量方法，收敛得较好。

优化方法和适用学习率的选择，取决于数据、模型和任务。当前场景下观察到的特性，在别的类似场景下，也值得尝试：

1. 先使用 Adadelta 方法得到一个收敛结果，作为基线指标。
2. 在不同的学习率下，用 Adam 方法靠拢和超越基线指标，找到较合适的初始学习率和新的基线指标。
3. 训练后期，递减学习率，并尝试其他优化方法（动量方法或 NAG），探索更好的收敛结果。

5.7 总体模型结构

构建一个和 Le-Net5 相近的模型，整个结构如图5.5所示。

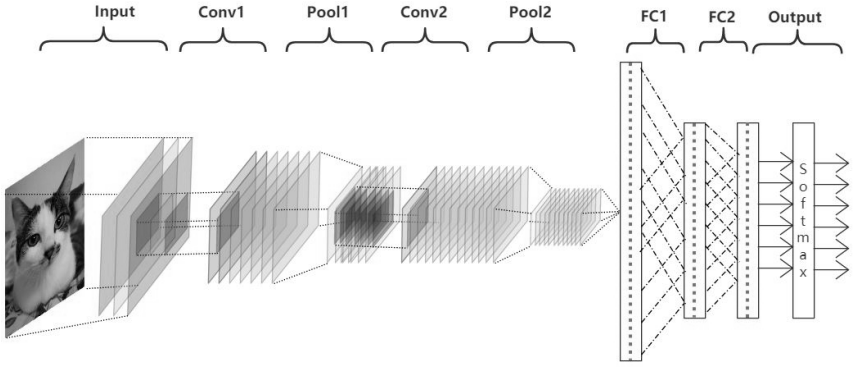

图 5.5 CNN 模型结构

输入数据维度为 28 × 28 的单通道灰度图片，之后的 4 层是两组卷积和池化单元，卷积层输出通过 ReLU 函数激活；接下来经过两个全连接层，得到分类结果，最后经过 Softmax 得到每个分类上的概率分布。

由于 MNIST 输入的数据是灰度图片，只有一个颜色通道，设置第一个卷积层输入通道为 1，有 32 个卷积核，即卷积层输出深度为 32；池化层改变数据的输入尺寸，不改变本层输入数据的深度，输出深度仍为 32；第二个卷积层输入通道为 32，有 64 个卷积核, 网络各层规格和超参数定义参见表5.3所示的定义。

表 5.3 卷积神经网络模型实现，各个处理层规格与超参数定义

No.	layer	input_size	channel	pad	strides	filters	Act
1	conv-1	28×28	1	2	1	5×5×32	ReLU
2	pool-1	28×28	32	0	2	2×2	\
3	conv-2	14×14	32	2	1	5×5×64	ReLU
4	pool-2	14×14	64	0	2	2×2	\
5	fc1	3136	\	\	\	\	ReLU
6	fc2	512	\	\	\	\	\
\	softmax	10	\	\	\	\	\

用这个六层结构的 CNN 模型，在尝试不同优化方法找到合适的学习率后，采用之前章节实现的 Adam 优化方法，进行各层参数的迭代更新。

5.8 使用 CNN 实现 MNIST 手写数字识别验证

这个**不借助深度学习框架**的 CNN 模型实现：卷积层、池化层，以及学习策略的优化方法，分别作为独立模块，封装为各自的实现类。也可以参照这个样例，复用源码中已实现的基础构件，自行配置 CNN 多层网络结构，观察模型训练的过程和结果。

使用这个具有两组 Conv-Pool 单元的卷积神经网络模型，在 MNIST 数据集上执行训练，每轮训练过程中，先后输出两次阶段性训练结果，以便观察训练过程中模型的收敛情况：

```
epoch 0, learning_rate= 0.00200000
epoch 0, loss=0.454550, loss_v=0.387560, acc=0.850, acc_v=0.891
epoch 1, learning_rate= 0.00100000
epoch 1, loss=0.038961, loss_v=0.070962, acc=0.990, acc_v=0.975
epoch 2, learning_rate= 0.00040000
epoch 2, loss=0.013229, loss_v=0.031253, acc=0.995, acc_v=0.988
epoch 3, learning_rate= 0.00020000
```

```
epoch 3, loss=0.028642, loss_v=0.028871, acc=0.985, acc_v
    =0.989
epoch 4, learning_rate= 0.00010000
epoch 4, loss=0.005758, loss_v=0.020398, acc=1.000, acc_v
    =0.992
...
```

经过前 30 次迭代训练，在验证集上正确率超过 80%；5 轮训练后，得到超过 99% 的验证正确率，如图5.6所示。

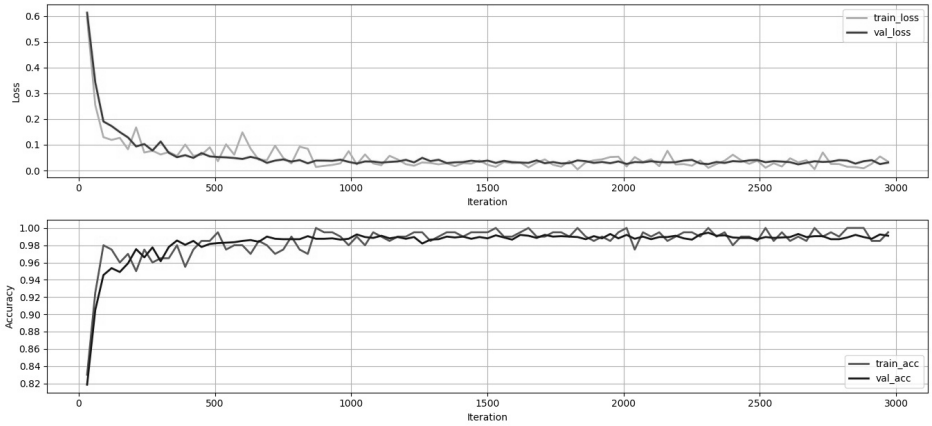

图 5.6　CNN 在 MNIST 数据集上的训练效果

5.9　小结

卷积神经网络是深度学习算法中最为重要的基础模型之一，其捕捉深层特征的强大能力，使之为公开数据集上的历届冠军算法提供了核心支撑。

本章对 CNN 模型基本构件的原始实现，做了算法的提速改进，然后讨论了深度学习的重要一环：学习策略的优化方法。对所讨论的各种方法，均提供了一种算法实现供读者参考，接着通过实验对各种优化方法的特点做了比较；最后，使用这个**不借助深度学习框架**的卷积神经网络模型，在 MNIST 上得到超过 99% 的分类正确率。

拓展阅读

斯坦福大学的卷积神经网络计算机视觉课程 cs231n，尝试用各种符合直觉的方式，讲解卷积神经网络的算法原理。

The Stanford CS class CS231n

`http://cs231n.stanford.edu`

参考文献

[1] John Duchi, Elad Hazan, and Yoram Singer. **Adaptive Subgradient Methods for Online Learning and Stochastic Optimization**. Journal of Machine Learning Research, 2011, 12:2121–2159.

[2] Diederik P Kingma, Jimmy Lei Ba. **Adam: a Method for Stochastic Optimization**. International Conference on Learning Representations, 2015:1–13.

[3] Sebastian Ruder. **An overview of gradient descent optimization algorithms.** arXiv:1609.04747v2, 2017.

[4] Zeiler. **ADADELTA: an adaptive Learning Rate method.** arXiv:1212.5701, 2012.

第 6 章

批量规范化（Batch Normalization）

"Wouldn't you like to know why reducing internal covariate shift speeds up gradient descent?
...
Wouldn't you like to know what internal covariate shift is?"

"你们不想知道为什么减少 ICS 就能加速 GD 过程吗？
. . . . . .
你们就不想知道何为 ICS 吗?"

Ali Rahimi@NIPS 2017

批量规范化（Batch Normalization）方法能大幅加速模型训练，同时保持预测正确率不降，因而被一些优秀模型采纳为标准模型层。

这一章，从深度神经网络的训练难题开始，描述批量规范化方法的初衷、算法、效果和原理分析，然后**不借助深度学习框架**实现这个算法，并在数据集上验证效果。

6.1 挑战：深度神经网络不易训练

从第一个神经网络：多分类模型到增加隐藏层的深度模型，再到引入卷积层与池化层处理的卷积神经网络，模型的处理层变得多样化，网络的结构也逐步加深了。

与最初的简单结构相比较，更复杂的模型能够捕捉到更丰富的数据特征，在 MNIST 数据集上的分类表现也越来越好。同时，模型也有了更多隐藏节点数和更大的参数量，如表6.1所示。

表 6.1 目前已实现的神经网络模型，结构从简到繁，在 MNIST 数据集上的验证正确率逐步提高

模型	结构	隐藏节点	参数量	正确率
单层全连接网络	无隐藏层	无	7850	>92%
两层全连接网络	一个隐藏层	512	407060	>98%
卷积神经网络	六个隐藏层	6044	>161 万	>99%

全连接层和卷积-池化单元，是卷积神经网络的基本结构。通过这些基本结构的组合与叠加，可以构造出各种各样的深度神经网络（Deep Neural Networks）模型。这些深度神经网络模型，往往需要在训练初始，从小到大尝试不同的学习率，观察收敛情况，训练过程中不断调整超参数的取值，逐渐找到较优的组合。此外，饱和非线性模型因其函数在饱和区导数趋于 0 故使模型不易训练。

为了应对这些挑战，**批量规范化方法（Batch Normalization, BN）**方法在 2015 年被提了出来。

6.2 批量规范化方法的初衷

批量规范化方法的作者把上述模型不易训练问题归因于 **Internal Covariate Shift**（ICS），认为：在深度神经网络模型的训练过程中，每一层输入数据的分布都会随前一层参数的变化而变化，不同处理层之间输入数据分布特征的变化，使模型的训练变得复杂，从而带来了上述问题。

然而，什么是“协变量（Covariate）”？什么是“偏移（Shift）”？这个“内

部协变量偏移（Internal Covariate Shift）”又是什么？

监督学习有个基本假设：源空间的训练数据和目标空间的待预测数据，被看成是按照同一个联合概率分布**独立同分布**产生的。有了这个基本假设，再来看通常所说的几种分布偏移的含义分别是什么。

6.2.1 数据集偏移

首先，训练数据和待预测数据分布不同的情况被称为**数据集偏移**（**Dataset Shift**）。

如果训练样本容量过小，或者实验设计不当，使训练样本的特征不能反映真实世界的数据特征，就会导致数据集偏移。

6.2.2 输入分布偏移

输入分布偏移是一种特殊的数据集偏移。

定义输入数据 x 是解释变量（explanatory variable）或协变量（covariate），类别标注 y 是应变量（response variable），如果训练数据和待预测数据的条件概率相同，而边缘概率不同。

$$P_{\mathrm{tr}}(y|x) = P_{\mathrm{te}}(y|x) \tag{6.1}$$

$$P_{\mathrm{tr}}(x) \neq P_{\mathrm{te}}(x) \tag{6.2}$$

这种情况被称为**输入分布偏移**。

又由条件概率与联合概率的关系：

$$\begin{cases} P_{\mathrm{tr}}(x,y) = P(y|x)P_{\mathrm{tr}}(x) \\ P_{\mathrm{te}}(x,y) = P(y|x)P_{\mathrm{te}}(x) \end{cases} \tag{6.3}$$

可以推知，此时训练数据和待预测数据的联合概率也不同：

$$P_{\mathrm{tr}}(x,y) \neq P_{\mathrm{te}}(x,y) \tag{6.4}$$

显然，训练出来的模型，推理预测是失准的。

6.2.3 内部偏移

BN 方法的作者用 ICS 来表述机器学习模型中层间输入数据分布变化的情况，然而没有给出 ICS 的正式定义。他希望通过减少层间协变量偏移，改善模型训练效率，以此为初衷，提出了 BN 方法。

BN 方法是否是通过减少 ICS 而改善训练效果的呢？方法提出后，有其他研究人员设计了实验，根据对实验结果的观察，提出了截然不同的观点，在进一步讨论这些观点之前，需要先了解 BN 方法具体的计算过程。

6.3 批量规范化的算法

BN 方法的前向计算，在训练和推理测试的时候，采取了不同的算法，反向传播也稍复杂一些；提出方法的原作者在论文中给出了方法的处理逻辑和算法表达式的结论，但没有详细给出反向传播的推导过程。下面分别来看。

6.3.1 训练时的前向计算

模型训练前向传播时，设每个容量为 m 的 mini-batch 是样本集合 X，集合中的样本是 x_i，其中 $(i = 1, 2, \cdots, m)$。首先计算集合中全部样本的算术均值。

$$\mu_{\text{batch}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{6.5}$$

再计算 mini-batch 集合的方差。

$$\sigma^2_{\text{batch}} = \frac{1}{m}\sum_{i-1}^{m}(x_i - \mu_{\text{batch}})^2 \tag{6.6}$$

接下来根据前两步得到的统计量，对集合中的每个单一样例 x 采用下面的步骤做批量规范化处理。

$$\hat{x}_i = \frac{x_i - \mu_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}} + \epsilon}} \tag{6.7}$$

$$y_i = \gamma\hat{x}_i + \beta \tag{6.8}$$

其中，式 (6.7) 分母的 ϵ（1e-8）是平滑项（smoothing term）, 用于在方差极小的情况下，避免表达式在数值计算时发生除 0 错。

观察 BN 方法前向训练的前三步，把输入的 mini-batch 批量数据，规范化为：**均值为 0、方差近似 1** 的标准化变量，再通过式 (6.8) 的线性变换，做伸缩（scaling）和平移（shifting），通过 γ 和 β 参数训练，学习合适的伸缩与平移幅度，恢复模型表达，得到 mini-batch 的批量规范化输出；训练完成后，学习得到 γ 和 β 参数，如图6.1所示。

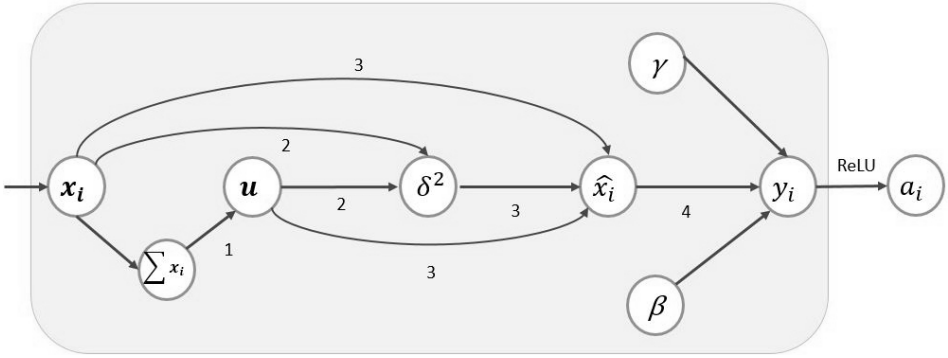

图 6.1　批量规范化训练时的前向计算；
先计算标准化变量，再把输入数据高斯化，
最后用训练得到的缩放和平移参数恢复特征表达

训练过程中，保留所有 mini-batch 计算得到的样本均值和方差，用于推理验证时做无偏估计。

你可以进一步了解**规范化**，也可以跳到卷积层的批量规范化处理一节继续阅读，不影响方法使用。

6.3.2　规范化与标准化变量

批量规范化的“规范化”是利用标准化变量把 mini-batch 中的输入转换为近似正态分布的步骤。

观察批量规范化方法前向训练的前三步：式 (6.5)、式 (6.6) 和式 (6.7)，可知转换后的期望：

$$\mathbb{E}(\hat{x}) = \frac{1}{\sqrt{\sigma^2_{\text{batch}} + \epsilon}} \mathbb{E}(x - \mu_{\text{batch}}) = \frac{1}{\sqrt{\sigma^2_{\text{batch}} + \epsilon}} [\mathbb{E}(x) - \mu_{\text{batch}}] = 0 \quad (6.9)$$

转换后的方差：

$$\begin{aligned}\mathrm{Var}(\hat{x}) &= \mathbb{E}\{[\hat{x}-\mathbb{E}(\hat{x})]^2\} \\ &= \mathbb{E}\{\hat{x}^2 - 2\hat{x}\mathbb{E}(\hat{x}) + [\mathbb{E}(\hat{x})]^2\} \\ &= \mathbb{E}(\hat{x}^2) - 2[\mathbb{E}(\hat{x})]^2 + [\mathbb{E}(\hat{x})]^2 \\ &= \mathbb{E}(\hat{x}^2) - [\mathbb{E}(\hat{x})]^2 \\ &= \mathbb{E}[(\frac{x-\mu_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}}+\epsilon}})^2] \\ &= \frac{1}{\sigma^2_{\text{batch}}+\epsilon}\mathbb{E}[(x-\mu_{\text{batch}})^2] \\ &= \frac{\sigma^2_{\text{batch}}}{\sigma^2_{\text{batch}}+\epsilon} \approx 1\end{aligned} \tag{6.10}$$

由于平滑因子 ϵ 取值较小，转换后的结果，均值为 0，方差接近 1。

BN 方法正是基于这一原理，通过训练时前向计算的前三步，把一个 mini-batch 的输入数据规范化为近似标准正态分布。

再观察式 (6.8)，前三步的输出，经过伸缩和平移变换之后，在这一步，使用学习得到的参数，恢复数据的特征表达；得到的结果均值为 b，而方差为 $\frac{\gamma^2\sigma^2}{\sigma^2+\epsilon} \approx (\gamma\sigma)^2$，二者都是常数，且方差大于 0，这使得最后输出的数据**仍然近似满足正态分布**。

需要注意的是，训练完成后，模型对于测试样本的推理预测，采用了不同的算法。

6.3.3 推理预测时的前向计算

验证预测时，待处理的样本量可能较小，甚至可能是单个样本，无从计算均值和方差统计量，因而**处理方式与训练时不同**，此时要使用训练时保存的所有批次样本均值和方差，而不是验证样本的均值和方差，与训练得到的 γ、β 参数对应计算。

先以训练时保存的所有批次样本的均值作为待预测样本的均值。

$$\mathbb{E}[x] = \mathbb{E}_{\text{batch}}[\mu_{\text{batch}}] \tag{6.11}$$

再用训练时保存的所有样本的方差，做待预测样本方差的无偏估计。

$$\mathrm{Var}[x] = \frac{m}{m-1}\mathbb{E}_{\mathrm{batch}}[\sigma^2_{\mathrm{batch}}] \tag{6.12}$$

最后用之前估计得到的统计量与训练得到的 γ、β 参数一起，进行样本的前向推理预测步骤。

$$\begin{aligned} y_{\mathrm{inf}} &= \frac{\gamma}{\sqrt{\mathrm{Var}[x]+\epsilon}}\cdot x + (\beta - \frac{\gamma\mathbb{E}[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}) \\ &= \gamma\cdot\frac{x-\mathbb{E}[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}} + \beta \qquad \blacksquare \end{aligned} \tag{6.13}$$

相对于容量较大的训练数据，把验证预测输入当作抽样样本，在式 (6.12) 中，用**无偏估计**来部分抵消批量规范化层处理对预测样本的影响，使输出仅由预测样本的特征来决定。可以证明，用无偏估计得到样本方差，和训练数据总体方差是一致的。

接下来讨论在全连接层和卷积层如何应用 BN 方法。

6.3.4 全连接层和卷积层的批量规范化处理

假设深度神经网络的第 l 层是一个线性处理的全连接层，权值参数和偏置项分别记作 $\boldsymbol{W}^{(l)}$ 和 $\boldsymbol{b}^{(l)}$，前一层的激活后输出是 $\boldsymbol{a}^{(l-1)}$，则本层原始输出 $\boldsymbol{a}^{(l-1)}$ 为

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\cdot\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}$$

如果对这个全连接层做批量规范化处理，则经过训练的 β 参数，已经将本层线性变换的结果 $\boldsymbol{a}^{(l-1)}$ 做了合适的平移，因此偏置项 $\boldsymbol{b}^{(l)}$ 可以忽略不用，搭配批量规范化处理的全连接层，可以不需要再定义和训练偏置项参数。

再看卷积层，把当前层输出展开为求和表达式。

$$z_{d,i,j} = \sum_{\mathrm{ch}=0}^{\mathrm{CH}-1}\sum_{r=0}^{R-1}\sum_{c=0}^{C-1} w_{d,\mathrm{ch},r,c}x_{\mathrm{ch},i+r,j+c} + b_d$$

如果把卷积层置于批量规范化层之前，同样可以无须定义和训练卷积运算的偏置项参数。

由于卷积层输出特征张量的深度，是该层卷积核的个数决定的，每个输出深度上的特征张量（feature map）不同区域使用同一组 γ、β 参数做批量规范化处理，因而在不同的输出深度上，批量规范化层需要分别定义 γ、β 参数各自训练。

6.4 批量规范化的效果

从结果看，深层神经网络使用 BN 普遍取得了更好的训练效果。

首先，缓解了**梯度传递问题**，使模型能适应更大的学习率，加速了训练；其次，改善了**饱和非线性模型**不易训练的问题；此外，还起到了**正则化**的作用。

6.4.1 梯度传递问题

对神经网络的全连接层，第 l 层（全连接层）原始输出：

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)} \cdot \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}$$

该层经过函数 f 激活后的输出：

$$\boldsymbol{a}^{(l)} = f(\boldsymbol{z}^{(l)})$$

第 $l-1$ 层的原始输出所对应的误差损失 $\boldsymbol{\delta}$ 可以由第 l 层的输出误差反推得到。

$$\boldsymbol{\delta}^{(l-1)} = \left((\boldsymbol{W}^{(l)})^{\mathrm{T}}\boldsymbol{\delta}_{fc}^{(l)}\right) \otimes f'(\boldsymbol{z}^{(l-1)}) \tag{6.14}$$

观察反向传播步骤可知，多层网络上每一层的反向传播误差会和各隐藏层转置后的权值参数矩阵做乘积运算（multiply）。

对于标量计算，如果用一个被乘数 a，不断累乘另外一个乘数 b, 最终的乘积 $abbbb\cdots$ 要么趋近于 $0(|b|<1)$, 要么膨胀到无穷大 $(|b|>1)$。

例如：

$$a \times 0.9^{100} \approx a2.66E-5 \qquad a \times 1.1^{100} \approx a1.38E4$$

深层网络反向传播时，梯度计算涉及各层权值参数矩阵的多次乘积运算，

权值参数矩阵的特征值或奇异值 (非方阵)，作用可以和上例中的标量乘数 b 做类比来理解，同样会影响梯度的缩放趋势。

在深度学习领域，反向传播梯度趋近于 0 和无穷大的情况，分别被称为**梯度弥散/消失（vanishing gradient）**和**梯度爆炸（exploding gradient）**。

再观察卷积层反向传播时误差的传递：

$$\boldsymbol{\delta}^{(l-1)} = \sum_{d=0}^{D} \boldsymbol{\delta}_d^{(l)} \odot \mathrm{rot180°}(W_d^{(l)}) \otimes f'(\boldsymbol{z}^{(l-1)})$$

各层的反向传播误差要与该层旋转后的卷积核做按位乘积（hadamard）运算，同样会产生梯度弥散和梯度爆炸问题。

应用 BN 方法后，各层传递给后一层的输出数据，以及反向传播给前一层的误差损失，都经过一次缩放调整，整个模型对学习率的选择和参数初始化的敏感度随之降低，从而改善了训练效果。

6.4.2 饱和非线性激活问题

结合激活函数的作用，只要再次观察激活函数的图像，如图6.2所示，就可以直观地理解饱和非线性激活问题：

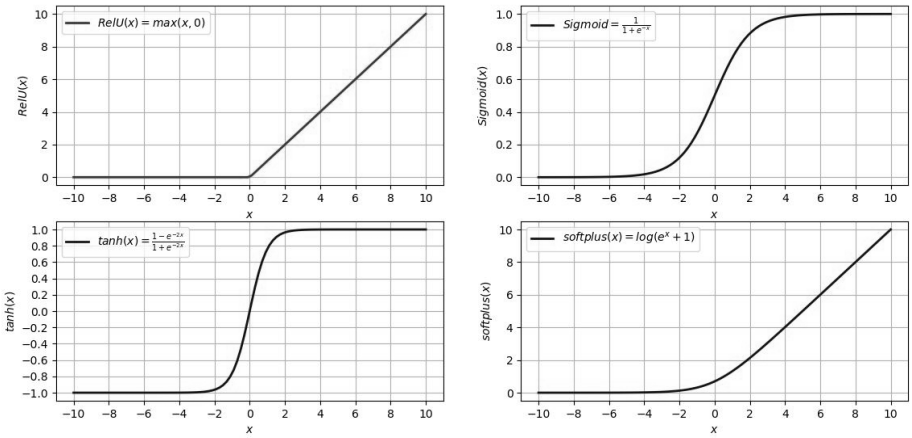

图 6.2　激活函数图像

饱和非线性（saturating nonlinearity）激活函数 **sigmoid** 和 **tanh**，在横轴 x=0 附近，导数比较大，而远离 x=0 区域的数据逐渐落入激活函数的饱和区：在两端导数较小的区域，激活后的神经元输出，对本层输入的变化不再敏

感，造成梯度消失，继而降低了训练速度。

非饱和（non-saturating）激活函数 **ReLU** 和 **softplus** 仅在单侧抑制输出，右侧有宽广的激活边界，可以有效地传递输入特征和反向梯度。

BN 方法经过规范化和缩放平移处理，可以使输入数据重新回到非饱和区，还可以更进一步地控制激活的饱和程度，以及非饱和函数对本层神经元输出进行抑制与激活的范围。

6.4.3 正则化效果

BN 方法的作者通过实验观察到：模型增加了 BN 处理，也有缓解过拟合的作用，对模型起到了正则化效果。

针对这一观察，作者分析了原因：由于批量规范化处理的机制，使一个 mini-batch 里的不同样本相互影响，起到了正则化作用。但是对这个判断着墨不多，显然，作者自己对这个原因分析并不满意。

2018 年，MIT 一个研究小组的工作认为：是**正则化**，而不是**去 ICS**，才是批量规范化有效的原因。

6.5 批量规范化为何有效

按照**去 ICS** 的思路：经过网络中前置层的变换，当前层输入的分布发生改变，影响了训练；批量规范化批量规范化方法之所以有效，是由于在非线性层之前，通过控制各层输入分布的均值和方差，来稳定各层输入的分布，从而促进了训练效果。

那么批量规范化是不是通过**去 ICS** 来改善了模型的呢？

或者反过来描述这个问题，如果在批量规范化之后**再度添加 ICS**，模型会变差吗？

研究小组在批量规范化层后增加随机噪声（random noise），这些随机噪声和输入数据是独立同分布的，然而不同于批量规范化处理后输出的数据分布，这些噪声均值不为 0，方差也不为 1；经过实验，训练出来的模型参数仍然好于不用 Batch Norm 的对照模型，由此证明方法的有效性和 ICS 不相关。

上面这个实验，可以说是这项工作的一大亮点。

如果和 ICS 不相关，那批量规范化为什么有效呢？

这个小组观察到，使用了 BN 方法的卷积神经网络模型和深度线性模型，误差损失和梯度都减少了抖动，优化损失的解空间更平滑，而一些同样能起到

平滑效果的正则化方法，在深度线性模型上，也起到了 Batch Norm 方法的优化效果。由此认为，BN 方法正是由于这些平滑效果，缓解了梯度传递与非饱和激活问题，从而提升训练性能的。

这个小组也试图探究 BN 方法**是否实现了去 ICS**，为此，计算了先后两次权值参数更新之间 W 梯度的变化，希望以此间接给出 ICS 的量化定义，通过实验观察和比较目标量的变化，认为批量规范化可能没有去除“ICS”，甚至可能使“ICS”加剧了。这个结论是小组从自己定义的“ICS”出发得到的，同样尚存争议。

6.6 批量规范化的反向传播算法

按照链式求导方法，根据后一层回传的误差损失可以计算初本层的批量规范化参数的梯度，以及向上层逆向传递的误差, 如图6.3所示。

已知 Loss 对 y_i 的偏导 $\frac{\partial L}{\partial y_i}$，求 $\frac{\partial L}{\partial \gamma}, \frac{\partial L}{\partial \beta}, \frac{\partial L}{\partial x_i}$。

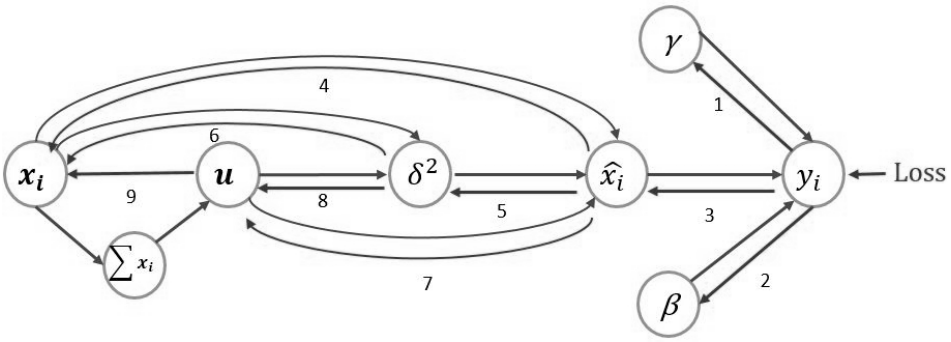

图 6.3 批量规范化反向传播

其中参数梯度：

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i} \cdot \hat{x}_i \tag{6.15}$$

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i} \tag{6.16}$$

在向上误差传递过程中，需要每次对 mini-batch 中的特定样本求偏导：

$$\begin{aligned}\frac{\partial L}{\partial x_i} &= \underbrace{\frac{\partial L}{\partial \widehat{xi}} \cdot \frac{\partial \hat{x}_i}{\partial xi}}_{\text{derivative of } \hat{x}_i} + \underbrace{\frac{\partial L}{\partial \mu} \cdot \frac{\partial \mu}{\partial x_i}}_{\text{derivative of } \mu} + \underbrace{\frac{\partial L}{\partial \sigma^2} \cdot \frac{\partial \sigma^2}{\partial x_i}}_{\text{derivative of } \sigma^2} \\ &= \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma^2+\epsilon}} + \frac{\partial L}{\partial \mu} \cdot \frac{1}{m} 2(x_i - \mu) + \frac{\partial L}{\partial \sigma^2} \frac{1}{m}\end{aligned} \tag{6.17}$$

分解来看其中的 $\frac{\partial L}{\partial \hat{x}_i}, \frac{\partial L}{\partial \mu}, \frac{\partial L}{\partial \sigma^2}$：

$$\frac{\partial L}{\partial \hat{x}_i} = \frac{\partial L}{\partial y_i} \cdot \gamma \tag{6.18}$$

$$\frac{\partial L}{\partial \mu} = \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial \mu} + \frac{\partial L}{\partial \sigma^2} \cdot \frac{\partial \sigma^2}{\partial \mu} \tag{6.19}$$

$$= \sum_{i=1}^{m} \frac{\partial L}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma^2+\epsilon}} + \frac{\partial L}{\partial \sigma^2} \cdot \frac{1}{m} \sum_{i=1}^{m} (-2)(x_i - \mu) \tag{6.20}$$

$$\frac{\partial L}{\partial \sigma^2} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i} \cdot \frac{\partial y_i}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial \sigma^2} \tag{6.21}$$

$$= \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i} \cdot (x_i - \mu) \cdot \frac{-(\sigma^2+\epsilon)^{-\frac{3}{2}}}{2} \tag{6.22}$$

将式 (6.18)、式 (6.20)、式 (6.22) 代入式 (6.17)，即可得到继续反向传递的误差损失。

式 (6.17) 与原作者论文（S. Ioffe *et al.*,2015）中反向传播的结果表达式是一致的，然而在实践的时候，还可以对式 (6.17) 提取公因式，进一步化简为

$$\frac{\partial L}{\partial x_i} = \frac{1}{m \cdot \sqrt{\sigma^2+\epsilon}} \left[m \frac{\partial L}{\partial \hat{x}_i} - \sum_{j=1}^{m} \frac{\partial L}{\partial \widehat{x}_j} - \widehat{x}_i \sum_{j=1}^{m} \frac{\partial L}{\partial \widehat{x}_j} \cdot \widehat{x}_j \right] \tag{6.23}$$

化简后的式 (6.23)，去除了原式 (6.17) 中冗余的中间变量，反向传播的运算结果虽然一样，但是在运算过程中能有效地提高内存使用效率。

6.7 算法实现

根据上一节的算法推导结果，本节将实现批量规范化的训练、反向传播、无偏估计预测的关键逻辑。

6.7.1 训练时的前向传播

在训练时，对每个 mini-batch 样本集合，首先计算均值、方差，然后得出该样本集合的标准化变量，再对集合中的样本逐个做缩放和平移。

$$\mu_{\text{batch}} = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma^2_{\text{batch}} = \frac{1}{m}\sum_{i-1}^{m}(x_i - \mu_{\text{batch}})^2$$

$$\hat{x}_i = \frac{x_i - \mu_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}} + \epsilon}}$$

$$y_i = \gamma\hat{x}_i + \beta$$

训练时前向传播计算的实现。

```
# 批量规范化训练时前向计算
def bnForward_tr(x, gamma, beta, eps):
    # 批次均值Mean
    mu = np.mean(x, axis=0)
    # 方差Var
    xmu = x - mu
    var = np.mean(xmu **2, axis = 0)
    #累计每个批次的Mean和Var，得到训练数据的统计量
    self.mov_avg_accum(mu, var)
    #高斯化
    ivar = 1./np.sqrt(var + eps)
    xhat = xmu * ivar
    #缩放和平移，仍然满足近似高斯分布
    out = gamma*xhat + beta
    #缓存中间结果，反向传播时复用
    cache = (xhat,gamma,ivar)

    return out, cache
```

在训练时，计算并缓存平滑标准差的倒数（reciprocal），反向传播的时候就可以直接复用，避免无谓消耗算力。

每个批次训练后，需要计算和保存各个批次均值和方差的**滑动平均结果**，

最后得到所有训练批次的滑动平均统计结果，用于测试推理的时候做无偏估计预测。

```
# 滑动平均累计mini-batch的mean和variance
def mov_avg_accum(self,mu,var):
    #设置滑动平均系数
    decay = self.decay_accum
    self.mu_accum = decay * self.mu_accum + (1-decay) * mu
    self.var_accum = decay * self.var_accum + (1-decay) *
        var
```

此处滑动平均系数 decay_accum 设置为 0.95。

6.7.2 反向传播

反向传播时，需要分别计算批量规范化层的**参数梯度和误差损失**。

首先计算缩放参数的梯度。

$$\frac{\partial L}{\partial \gamma}=\sum_{i=1}^{m}\frac{\partial L}{\partial y_i}\cdot \hat{x}_i$$

其中 $\hat{x}_i$ 用待预测样本的无偏估计统计量计算得出的标准化因子。

再计算平移参数的梯度。

$$\frac{\partial L}{\partial \beta}=\sum_{i=1}^{m}\frac{\partial L}{\partial y_i}$$

由于结构中批量规范化层之前还有卷积层，需要计算误差损失进行反向回传，所以用提取公因式之后的最终表达式计算误差损失：

$$\frac{\partial L}{\partial x_i}=\frac{1}{m\cdot\sqrt{\sigma^2+\epsilon}}\left[m\frac{\partial L}{\partial \hat{x}_i}-\sum_{j=1}^{m}\frac{\partial L}{\partial \widehat{x}_j}-\widehat{x}_i\sum_{j=1}^{m}\frac{\partial L}{\partial \widehat{x}_j}\cdot\widehat{x}_j\right]$$

根据推导结果，实现批量规范化反向传播。

```
#批量规范化反向传播
def bnBackward(dout, cache):
    # 复用前向训练保留的中间结果
```

```
    xhat,gamma,ivar = cache
    # mini-batch样本数
    M = dout.shape[0]
    #参数梯度计算
    dbeta = np.sum(dout, axis=0)
    dgamma = np.sum(dout*xhat, axis=0)
    #计算向上传递的误差损失
    dxhat = dout * gamma
    dx = 1./M* ivar * (M*dxhat-np.sum(dxhat, axis=0) -
        xhat*np.sum(dxhat*xhat,axis=0))

    return dx, dgamma, dbeta
```

批量规范化的反向传播推导较为复杂，然而推导出结果，并进一步化简以后，算法的实现却可以十分简洁。

6.7.3 推理预测

和之前讨论的基础核心算法相比，BN 方法最大的不同在于推理预测时，需要基于对统计量的无偏估计来完成前向计算。

这就需要用到训练时前向计算所保留的所有训练批次均值与方差的累计滑动平均结果，对验证样本做**无偏估计预测**。

$$\mathbb{E}[x] = \mathbb{E}_{\text{batch}}[\mu_{\text{batch}}]$$

$$\text{Var}[x] = \frac{m}{m-1}\mathbb{E}_{\text{batch}}[\sigma^2_{\text{batch}}]$$

$$y_{\text{inf}} = \gamma \cdot \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} + \beta$$

实现批量规范化的无偏估计推理预测：

```
# 批量规范化推理预测
def bnForward_inf(self, x, gamma, beta, eps):
```

```
    M = self.MiniBatchesSize
    # 预测样本的无偏估计方差
    var = M / (M - 1) * self.var_accum
    # 推理结果
    tx = (x - self.mu_accum) /np.sqrt(var + eps)
    out = gamma * tx + beta
    # 返回BN层推理预测时的前向计算输出
    return out
```

以上源码，实现了批量规范化的关键算法，接下来把 BN 方法应用在之前实现的卷积神经网络模型中，观察训练的效果。

6.8 调整学习率和总体结构

6.8.1 模型结构

批量规范化层在模型结构中的合适位置，需要视数据、任务和模型结构来尝试确定，通常可以把 BN 层放在卷积层或者全连接层之后，激活函数之前。

在本章实现的原卷积神经网络模型中，在两个卷积层之后，分别添加批量规范化层再做 ReLU 激活输出，增加了两处批量规范化处理之后，模型结构成为图6.4。

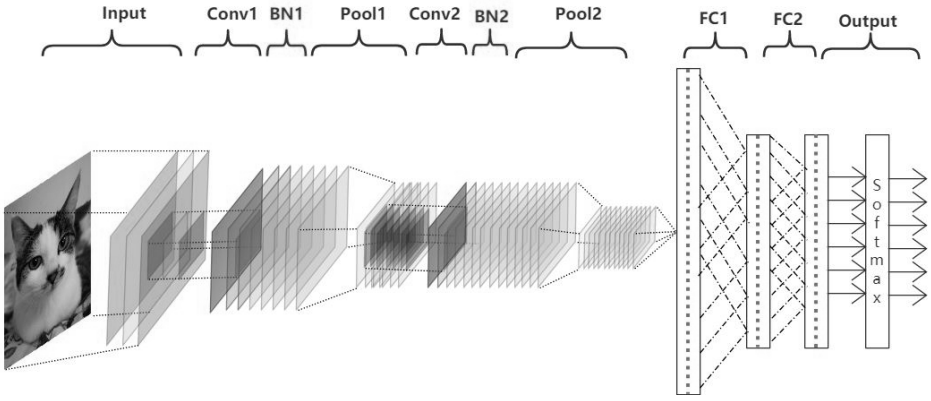

图 6.4 在卷积神经网络模型中增加批量规范化层

6.8.2 卷积层批量规范化的实现

全连接层输出只有 mini-batch（N）和样本特征（D）两个维度。而卷积层输出除了增加颜色通道（C）维度以外，每个样本的特征还包含了宽（W）和高（H）维度。在卷积层做批量规范化处理时，由于每个输出深度上的特征张量（feature map）不同区域使用同一组 γ 和 β 参数进行训练和推理预测；因此，只要把卷积层的输出数据在 mini-batch（N）、宽（W）和高（H）维度上做合并，即可复用已实现的批量规范化方法，稍加改造来实现卷积层批量规范化的前向传播计算。计算完成之后，先恢复卷积层输出数据的原始维度，再进行神经网络中后一环节的处理。

```
# 先对卷积层原始输出进行维度合并
if len(xOri.shape) == 4:
    N,C,H,W = xOri.shape
    x = xOri.transpose(0, 2, 3, 1).reshape(N * H * W, C)
    elif len(xOri.shape) == 2:
    N,D = xOri.shape
    x = xOri

# 再做批量规范化前向计算
# 复用批量规范化前向传播算法
...
# 最后恢复原始维度
if len(xOri.shape) == 4:
    outOri = out.reshape(N, H, W, C).transpose(0, 3, 1, 2)
```

卷积层批量规范化的反向传播，同样先做维度合并，再复用已实现的批量规范化反向传播算法。最后将回传误差的恢复成后一层原始回传误差的维度，再继续反向传播至前一层。

```
# 先对卷积层原始回传误差损失进行维度合并
if len(doutOri.shape) == 4:
    N, C, H, W = doutOri.shape
    dout = doutOri.transpose( 0, 2, 3, 1 ).reshape( N * H
        * W, C)
# 再做批量规范化的反向传播计算
# 复用批量规范化反向传播算法
```

```
...
# 最后恢复回传误差的原始维度
if len(doutOri.shape) == 4:
    dxOri = dx.reshape(N, H, W, C).transpose(0, 3, 1, 2)
```

上述实现复用了全连接层的批量规范化算法，通过维度转换简洁地实现了卷积层批量规范化的前向和反向传播运算。

6.8.3 引入批量规范化后的递减学习率

由于 BN 方法能使参数优化变得更平滑从而加速训练，在模型中添加批量规范化层后，可以调整学习率，每次迭代后用更大的步长更新各处理层的训练参数。

首先，在 LeNet-5 每个递增层级的基础上，把表5.2的学习率放大一倍，如表6.2所示。

表 6.2 使用批量规范化方法后的递减学习率参数，在本章任务场景下放大一倍

训练轮数	原学习率（无 BN 层）	学习率（有 BN 层）
0-1	0.005	0.01
2-4	0.002	0.004
5-7	0.000 1	0.000 2
8-11	0.000 05	0.000 1
12-thereafter	0.000 01	0.000 02

学习率放大的倍数同样视数据、任务和模型结构而定，需要一边观察模型的收敛情况，一边调整参数。在当前场景下，经过实验观察，把每个递增层级上的学习率在原基础上放大一倍效果较好。这一系列学习率被全局应用于各层权参、偏置项。

整个模型各的各处理层参数的更新，包括卷积层后新增的两层批量规范化所含的缩放和平移参数 γ 和 β，复用已讨论实现的 **Adam 优化算法**。

6.9 在 MNIST 数据集上验证结果

为了观察对比批量规范化对模型的提升效果，仍然在 MNIST 数据集上用第 5 章准备好的预处理数据，做手写数字识别验证，观察这个改进模型的收敛过程如图6.5所示。

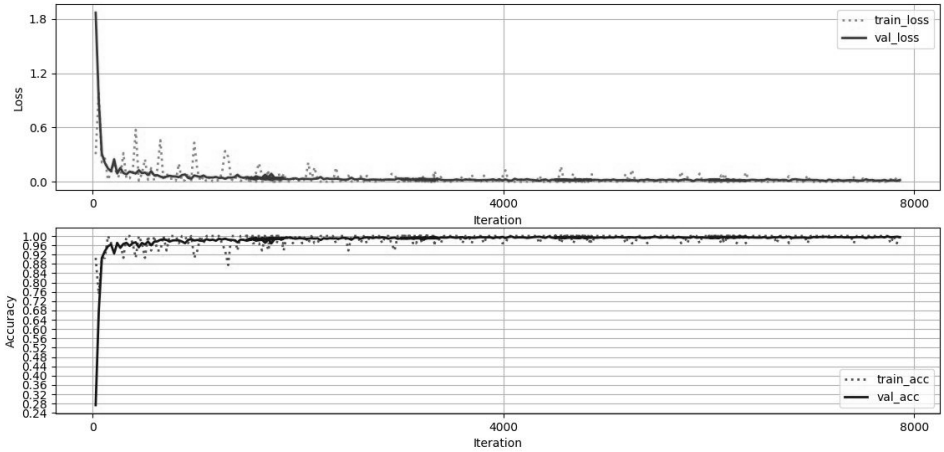

图 6.5 使用 BN 后的模型指标曲线;
上图：损失曲线，下图：Accuracy 曲线

在模型首轮训练的初期迭代过程中，验证数据集上推理预测偏差比较大，之后正确率才能快速提高；是因为在训练的前期，少量训练样本均值、无偏估计方差的滑动平均结果与总体数据集特征的差异仍然较大。

首轮训练到了中期的迭代批次，损失曲线仍然出现了幅度明显的波动，表明在随机抽取的 mini-batch 上产生的局部规范化，得到的标准化变量存在上下起伏，仍然会影响 γ 和 β 参数的学习。此后，累计均值和无偏估计方差的滑动平均结果，趋于总体数据集的特征，批量规范化的优势逐渐显现出来。

之后的迭代批次，添加批量规范化处理的模型，稳定性和正确率明显优于相同学习率的对照模型，经过 5 轮迭代后，独立测试数据集上的推理预测正确率，稳定提升到 99.4% 以上。

6.10 小结

> "Sticking to a set of methods just because you can do theory about it, while ignoring a set of methods that empirically work better just because you don't (yet) understand them theoretically is akin to looking for your lost car keys under the street light knowing you lost them someplace else."
>
> "止步于已知领域，无视被实证的更好方法，就像明知道钥匙丢在了别处，却固守在路灯下寻找。"
>
> Yann LeCun's response to Ali Rahimi，2017

BN 方法的初衷是解决 ICS 问题。然而真正发挥的作用，是缓解了梯度传递问题和饱和非线性激活问题，通过平滑优化解空间起到了正则化作用，使模型对大步长学习率敏感度降低，更加易于训练。

尽管 BN 方法收效的原因尚在探索当中，但在实践中，确实能有效地加速收敛，因而被普遍使用在深层神经网络模型中，成为深度学习方法链条上重要的一环，被用于一系列 state-of-art 模型，也推动了理论研究向前迈进。

拓展阅读

宾夕法尼亚州立大学的概率论（STAT 414）和数理统计（STAT 415）课程，对本章涉及的数据高斯化、无偏估计有更进一步的介绍和计算举例。

The Pennsylvania State University STAT 414/415

`https://onlinecourses.science.psu.edu/stat414`

参 考 文 献

[1] S. Ioffe and C. Szegedy. **Batch normalization: Accelerating deep network training by reducing internal covariate shift.** arXiv:1502.03167, 2015.

[2] Arthur Gretton, Alexander J Smola, Jiayuan Huang, Marcel Schmittfull, Karsten M Borgwardt, and Bernhard Schölkopf. **Covariate shift by kernel mean matching.** 2009.

[3] Shimodaira H . **Improving predictive inference under covariate shift by weighting the log-likelihood function**. Journal of Statistical Planning and Inference, 2000, 90(2):227-244.

[4] Ali Rahimi, Ben Recht. **Back when we were kids.** In NIPS Test-of-Time Award Talk, 2017.

[5] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, Aleksander Madry. **How does batch normalization help optimization? (no, it is not about internal covariate shift).** arXiv:1805.11604, 2018.

[6] Ba Jimmy Lei, Kiros Jamie Ryan, Hinton Geoffrey E. **Layer Normalization**. 2016.

[7] Wu Yuxin, He Kaiming. **Group Normalization**.The European Conference on Computer Vision (ECCV), 2018:3-19.

第 7 章

循环神经网络（Vanilla RNN）

> “In the model network … Memories are retained as stable entities and can be correctly recalled … time ordering of memories can also be encoded …”
>
> “这个模型网络里 …… 记忆是可以存取的实体，…… 记忆的时间线也能被编码 ……”
>
> John Joseph Hopfield @Caltech , 1982

循环神经网络（Recurrent Neuron Network，RNN）是用于序列数据分析的模型，有着广泛的应用场景：

- 图像描述（image caption）。

- 语音识别与机器翻译。
- 以特定艺术风格写诗、作曲。
- 拟合远期资产价格曲线，试算折现盈亏。
- 根据社交媒体数据的情感特征分析市场情绪和大众预期。

  ……

RNN 是深度学习算法的核心构件，**为了更好地理解算法**，这一章从动机、结构到反向传播和学习策略，逐步分析，然后**不借助深度学习框架**，实现 RNN 模型，再应用于时序数据的分析预测，验证这个模型。

7.1 第一个挑战：序列特征的捕捉

之前描述的全连接神经网络（FCN）和卷积神经网络（CNN），其目标数据的样本是不分先后的；然而在真实世界中，样本之间可能存在序列关系，例如，样本排列的相对位置，或者样本序列在时间上的先后时序关系。为了分析这类场景下数据的特征，模型需要能够参考前序信息来支持当前决策。

比如下面这句话：

> 周六早上难得好天气，正好可以（去打球 / 睡懒觉）。

结合上文，既然“天气不错”，那么做出“去打球”这个决策的可能性应该高于“睡懒觉”，反之亦然。

为了获取这一类序列特征，**John Joseph Hopfield** 在 1982 年提出了支持记忆机制的霍普菲尔德网络（Hopfield Network）。

以此为基础，具备记忆机制的循环网络模型逐渐演进到现在，成为更有效的循环神经网络模型（RNN）。

7.2 循环神经网络的结构

RNN 的结构同全连接神经网络及卷积神经网络都有较大的差异。经过多年发展，RNN 模型又衍生出各式各样的变体，先从了解基本的单层模型入手。

7.2.1 单层 RNN

仍然用前面的例句，如果暂不考虑按照语义对句子做的分词处理，而是直接把每个单字语素拆分开来，整个句子可以看成是由多个语素依次排成的输入

序列：

[周, 六, 早, 上, 难, 得, 好, 天, 气, 正, 好, 可, 以,$\cdots$]

每个语素可以被看成时间序列 $[x_1, x_2, \cdots, x_t]$ 中的单个样本，RNN 模型顺序接受单个样本输入，处理后得到预测输出结果 y：“去打球”或者“睡懒觉”，如图7.1所示。

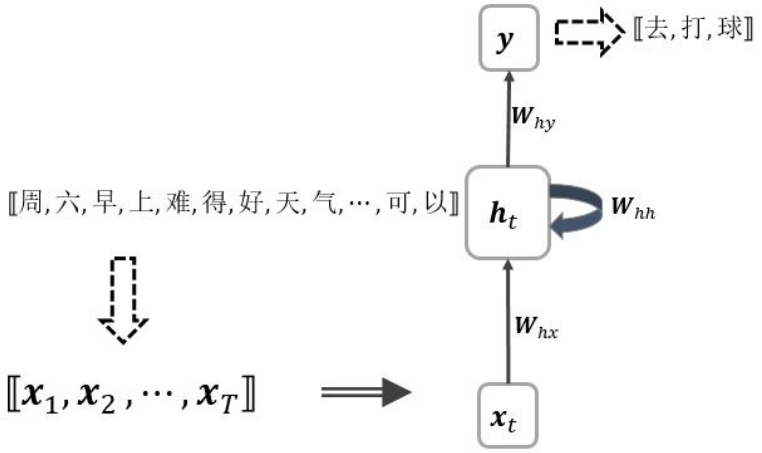

图 7.1　单层 RNN 结构，输入序列依次送入 RNN 隐状态节点进行运算，最后得到推理预测输出

这个单层结构中的 $\boldsymbol{h}_t$ 代表在时刻 t 的隐状态节点，$\boldsymbol{W}_{hx}$ ，$\boldsymbol{W}_{hh}$，$\boldsymbol{W}_{hy}$ 分别是前向传播计算时各个方向上使用的权值参数，每个节点上计算得到的原始输入使用函数 f 激活；在单层 RNN 上，各个时间步用样本对应当前输入序列位置的元素，和前一时间步的输出结果一起计算当前时间步隐藏节点的输出，作为下一个时间步的横向输入，继续参与计算。这个过程可以写为下面的计算表达式：

$$
\begin{aligned}
\boldsymbol{h}_1 &= f(\boldsymbol{W}_{hx}\boldsymbol{x}_1 + \boldsymbol{W}_{hh}\boldsymbol{h}_0 + \boldsymbol{b}) \\
\boldsymbol{h}_2 &= f(\boldsymbol{W}_{hx}\boldsymbol{x}_2 + \boldsymbol{W}_{hh}\boldsymbol{h}_0 + \boldsymbol{b}) \\
&\vdots \\
\boldsymbol{h}_t &= f(\boldsymbol{W}_{hx}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}) \\
\boldsymbol{y}_t &= \boldsymbol{W}_{hy}\boldsymbol{h}_t + \boldsymbol{b}_y \qquad \blacksquare
\end{aligned}
$$

RNN 模型结构也可以按照时间维度展开隐藏节点，用图7.2所示的单层

RNN 结构来等价表示。

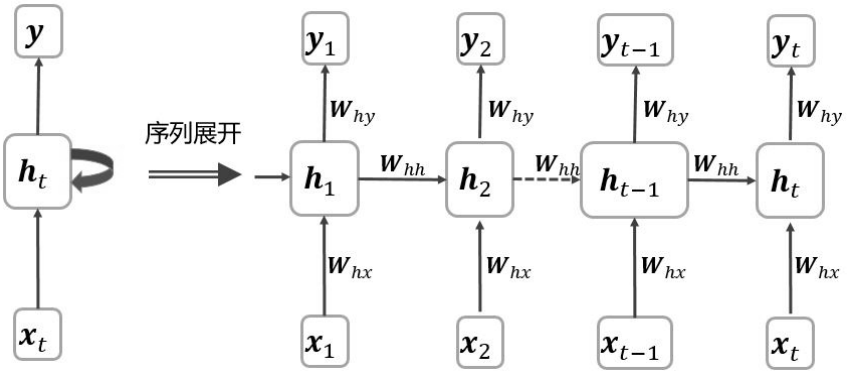

图 7.2　按序列展开单层 RNN 之后的结构，
可以更清晰地观察序列每一步的运算处理

三个权值参 $\boldsymbol{W}_{hx}$ ，$\boldsymbol{W}_{hh}$，$\boldsymbol{W}_{hy}$ 横向展开后，在当前层各个时间步上是共享的，参数总量取决于单个时间步上的输入维度，以及每个时间步上的隐状态节点维度，和时间步的序列长度无关。

7.2.2　双向 RNN

很多时候，也会遇到由下文反过来推理上文的场景，例如：

周六早上难得好天气, 正好可以（去打球 / 睡懒觉）,可别叫醒我。

综合上下文，既然有“别叫醒我”跟在后面，显然“睡懒觉”的可能性大幅提高。

在这一类需要结合上下文，在正序、倒序两个方向上做推理的场景中，可以通过**双向 RNN** 结构，加入**逆向推理**机制, 如图7.3所示。同正方向序列上的权值参数一样，逆序列的权参 $\boldsymbol{W}'_{hx}$ ，$\boldsymbol{W}'_{hh}$，$\boldsymbol{W}'_{hy}$ 在同一层 RNN 结构每个时间步上也是共享的。

$$
\begin{aligned}
\boldsymbol{h}'_t &= f(\boldsymbol{W}'_{hx}\boldsymbol{x}_t + \boldsymbol{W}'_{hh}\boldsymbol{h}'_{t+1} + \boldsymbol{b}') \\
&\vdots \\
\boldsymbol{h}'_2 &= f(\boldsymbol{W}'_{hx}\boldsymbol{x}_2 + \boldsymbol{W}'_{hh}\boldsymbol{h}_1 + \boldsymbol{b}) \\
\boldsymbol{h}'_1 &= f(\boldsymbol{W}'_{hx}\boldsymbol{x}_1 + \boldsymbol{W}'_{hh}\boldsymbol{h}'_0 + \boldsymbol{b}') \\
\boldsymbol{y}_t &= f(\boldsymbol{W}'_{hy}\boldsymbol{h}'_t + \boldsymbol{W}_{hy}\boldsymbol{h}_t + \boldsymbol{b}_y) \qquad \blacksquare
\end{aligned}
$$

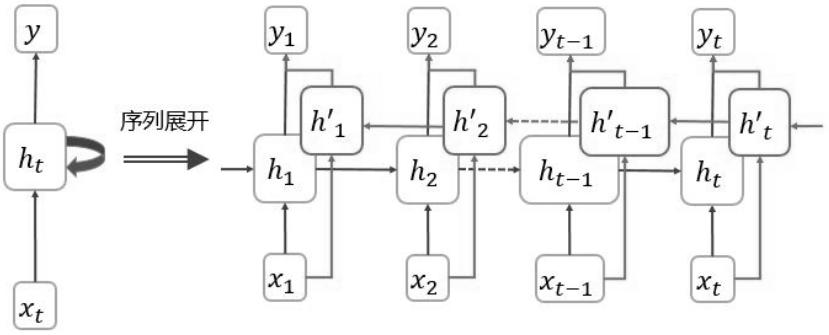

图 7.3 双向 RNN 的结构，增加了逆序特征的捕捉机制

双向 RNN 的参数量和隐藏节点个数扩展到单向模型的两倍，能通过捕捉输入数据的逆序特征来表达更深层的序列信息。

7.2.3 多层 RNN

如果遇到更复杂的序列信息，还有什么更进一步的措施来捕捉更丰富的数据特征呢？

回顾 CNN 卷积神经网络模型，可以增加**卷积核**个数或者增加 **Conv-Pool 单元**的数量，用更深的卷积过滤器，以及层级更多的网络处理层来捕捉更丰富的特征。

与之对应，RNN 的多层模型，可以通过**组合叠加更多的隐藏层**来抽取深层的序列特征，如图7.4所示。

在多层 RNN 结构中，每个隐藏层，各个方向上共享权值参数 $\boldsymbol{W}$。

$$
\begin{aligned}
\boldsymbol{h}_1 &= f(\boldsymbol{W}_{hx}\boldsymbol{x}_1 + \boldsymbol{W}_{hh}\boldsymbol{h}_0 + \boldsymbol{b}) \\
&\vdots \\
\boldsymbol{h}'_1 &= f(\boldsymbol{W}'_{hx}\boldsymbol{h}_1 + \boldsymbol{W}'_{hh}\boldsymbol{h}'_0 + \boldsymbol{b}') \\
&\vdots \\
\boldsymbol{h}''_t &= f(\boldsymbol{W}''_{hx}\boldsymbol{h}'_t + \boldsymbol{W}''_{hh}\boldsymbol{h}''_{t-1} + \boldsymbol{b}'') \\
\boldsymbol{y}_t &= \boldsymbol{W}_{hy}\boldsymbol{h}''_t + \boldsymbol{b}_y \qquad \blacksquare
\end{aligned}
$$

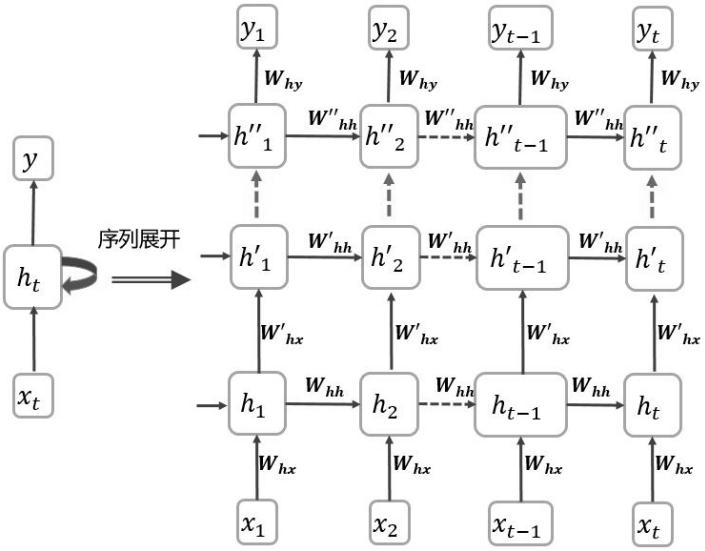

图 7.4　组合多层的 RNN 模型，捕捉更深层的序列信息

为方便反向传播推导，前面列举的 RNN 算法步骤，可以写为更一般的前向传播算法表达式。

7.3　RNN 前向传播算法

设 RNN 模型的输入 $\boldsymbol{x}$ 是 D 维向量，$\boldsymbol{W}$ 是权值参数矩阵，在任意时刻 t，隐藏节点的原始输出是该时刻输入 $\boldsymbol{x}_t$ 和 $t-1$ 时刻隐层节点输出的加权和：

$$\boldsymbol{z}_t = \boldsymbol{W}_{hx}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b} \tag{7.1}$$

如果选用双曲正切函数 tanh 作为节点输出的激活函数，则隐藏节点激活后输出为

$$\boldsymbol{h}_t = \tanh(\boldsymbol{z}_t) \tag{7.2}$$

对 RNN 多层模型，第 l 层 t 时刻隐藏节点原始输出为

$$\boldsymbol{z}_t^l = \boldsymbol{W}_{hx}^l \boldsymbol{h}_t^{l-1} + \boldsymbol{W}_{hh}^l \boldsymbol{h}_{t-1} + \boldsymbol{b}^l \tag{7.3}$$

整个 RNN 单元在最后一层 L 的输出为

$$\boldsymbol{y}_t = f(\boldsymbol{W}_{hy}\boldsymbol{h}_t^L + \boldsymbol{b}_y^L) \qquad \blacksquare \tag{7.4}$$

以上是 Vanilla RNN 前向计算的全部表达式。

RNN 的算法表达式中，也可以把 $\boldsymbol{W}_{hx}$ 和 $\boldsymbol{W}_{hh}$ 合二为一，拼接成一个 $\boldsymbol{W}$ 权值参数矩阵；相应把上一层输入和前一时刻隐藏节点输出也拼接在一起，成为 $[\boldsymbol{x}_t, \boldsymbol{h}_{t-1}]$ ，再和权参 $\boldsymbol{W}$ 做仿射变换：

$$\boldsymbol{z}_t = \boldsymbol{W}[\boldsymbol{x}_t, \boldsymbol{h}_{t-1}] + \boldsymbol{b} \tag{7.5}$$

训练和预测时，只要表达式保持一致，两种方法的结果是等价的。这里采用权参分开来书写表达式，是为了便于观察反向传播算法的推导步骤。

7.4 RNN 反向传播算法

与 CNN 的相比，RNN 的反向传播算法多了一条按照时间反向传导的通道，也被称为 **BPTT**（Back-Propagation Through Time）**算法**。下面分别来看误差的反向传播和参数梯度的计算。

7.4.1 误差的反向传播

任意时刻 t，第 l 层隐藏节点的输出经过激活之后为

$$\boldsymbol{h}_t^l = f(\boldsymbol{z}_t^l) = \tanh(W_{hx}^l \boldsymbol{h}_t^{l-1} + \boldsymbol{W}_{hh}^l \boldsymbol{h}_{t-1} + \boldsymbol{b}^l) \tag{7.6}$$

误差反向传递到隐藏节点原始输出这个环节时，其误差 E 来自层级和时间两个方向，因此可由复合求导法则推得这个误差：

$$\boldsymbol{\delta}\boldsymbol{z}_t = \frac{\mathrm{d}E}{\mathrm{d}\boldsymbol{z}_t} = (\boldsymbol{\delta}\boldsymbol{h}_t + \boldsymbol{\delta}\boldsymbol{h}_t^{(l+1)}) \otimes \mathrm{diag}(\tanh'(\boldsymbol{z}_t)) \tag{7.7}$$

式 (7.7) 中 $\mathrm{diag}(\boldsymbol{f})$ 表示由向量 $\boldsymbol{f}$ 构成的对角阵，其中 $\boldsymbol{f} = \tanh'(\boldsymbol{z}_t)$。矩阵的 $\otimes$ 运算是之前章节介绍过的 Hadamard 逐元素乘积运算。

观察式 (7.7) 第二项，含有双曲正切函数的导函数。

7.4.2 激活函数的导函数和参数梯度

由激活函数的原函数：

$$\tanh\ z = y = \frac{1-\mathrm{e}^{-2z}}{1+\mathrm{e}^{-2z}} = \frac{\mathrm{e}^{2z}-1}{\mathrm{e}^{2z}+1} \tag{7.8}$$

可推得激活函数的导函数：

$$\begin{aligned}\frac{\mathrm{d}\tanh\ z}{\mathrm{d}z} &= \frac{\mathrm{d}(\frac{\mathrm{e}^{2z}-1}{\mathrm{e}^{2z}+1})}{\mathrm{d}z} = \frac{4\mathrm{e}^{2z}}{(\mathrm{e}^{2z}+1)^2} \\ &= \frac{(\mathrm{e}^{2z}+1)^2-(\mathrm{e}^{2z}-1)^2}{(\mathrm{e}^{2z}+1)^2} \\ &= 1-(\frac{\mathrm{e}^{2z}-1}{\mathrm{e}^{2z}+1})^2 \\ &= 1-(\tanh\ z)^2\end{aligned} \tag{7.9}$$

所以得到双曲正切函数一个有趣的性质：

$$\begin{aligned}\tanh\ z &= y \\ \tanh'\ z &= 1-y^2\end{aligned} \tag{7.10}$$

这是一个很好的性质：前向传播时，保留原始激活输出的数值结果，反向传播计算时，可以直接复用。

有了隐藏节点的误差，容易推导沿时间和层次继续传递的误差项：

$$\boldsymbol{\delta h}_{t-1} = \boldsymbol{W}_{hh}^{\mathrm{T}}\boldsymbol{\delta z}_t \tag{7.11}$$

$$\boldsymbol{\delta h}_t^{l-1} = \boldsymbol{W}_{hx}^{\mathrm{T}}\boldsymbol{\delta z}_t^l \tag{7.12}$$

在时间步方向上，权参 $\boldsymbol{W}_{hh}$ 在 t 时刻隐藏节点的梯度计算，类似全连接神经网络的反向传播梯度计算，在时刻 t 这个时间步上，用回传误差对时间步方向上的权值参数矩阵 $\boldsymbol{W}_{hh}$ 计算偏导数。

$$\nabla_{\boldsymbol{W}_{hh,t}}E = \frac{\partial E}{\partial \boldsymbol{z}_t}\frac{\partial \boldsymbol{z}_t}{\partial \boldsymbol{W}_{hh,t}} = \boldsymbol{\delta z}_t\frac{\partial(\boldsymbol{W}_{hx}\boldsymbol{x}_t+\boldsymbol{W}_{hh}\boldsymbol{h}_{t-1}+\boldsymbol{b})}{\partial \boldsymbol{W}_{hh,t}} = \boldsymbol{\delta z}_t(\boldsymbol{h}_{t-1})^{\mathrm{T}} \tag{7.13}$$

用同样的方法，可以推导出在层次方向上，权值参数矩阵 $\boldsymbol{W}_{hx}$ 在 t 时刻隐藏节点处的参数更新梯度：

$$\nabla_{\boldsymbol{W}_{hx,t}} E = \frac{\partial E}{\partial \boldsymbol{z}_t} \frac{\partial \boldsymbol{z}_t}{\partial \boldsymbol{W}_{hh,t}} = \boldsymbol{\delta}\boldsymbol{z}_t \frac{\partial (\boldsymbol{W}_{hx}\boldsymbol{h}_t^{l-1} + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b})}{\partial \boldsymbol{W}_{hh,t}} = \boldsymbol{\delta}\boldsymbol{z}_t (\boldsymbol{h}_t^{l-1})^{\mathrm{T}} \tag{7.14}$$

以及偏置项 $\boldsymbol{b}$ 在 t 时刻节点的参数更新梯度：

$$\nabla_{\boldsymbol{b}_t} E = \frac{\partial E}{\partial \boldsymbol{z}_t} \frac{\partial \boldsymbol{z}_t}{\partial \boldsymbol{b}_t} = \boldsymbol{\delta}\boldsymbol{z}_t \tag{7.15}$$

最后，把同层各个时刻隐藏节点处的参数梯度加总，就可以得到多层 RNN 模型结构中，任意一层权值参数矩阵和偏置项的反向传播梯度：

$$\nabla_{\boldsymbol{W}} E = \sum_{t=1}^{T} \nabla_{\boldsymbol{W}_t} E \tag{7.16}$$

$$\nabla_{\boldsymbol{b}} E = \sum_{t=1}^{T} \nabla_{\boldsymbol{b}_t} E \qquad \blacksquare \tag{7.17}$$

至此，完成了多层 RNN 模型中，BPTT 反向传播算法的全部误差传递，以及参数的更新梯度计算推导。

虽然主流的深度学习框架通过自动微分支持了反向传播，但是工程实践中，我们还是有必要了解原理，首要原因是，反向传播会有**抽象泄漏**（**Leaky Abstractions**）问题。

RNN 模型的基本算法并不复杂。然而，要根据数据场景，适当地完成长程时间步下的模型训练却不容易。BPTT 算法是基于梯度传递（gradient based）的算法，模型训练会面临梯度沿时间步的长程传递问题。

7.5 第二个挑战：循环神经网络的梯度传递问题

RNN 模型在同一层上共享权值参数，依时间步循环展开迭代计算，回顾 RNN 误差沿时间步的反向传播表达式：

$$\boldsymbol{\delta}\boldsymbol{h}_{t-1} = \boldsymbol{W}_{hh}^{\mathrm{T}} \boldsymbol{\delta}\boldsymbol{z}_t$$

观察反向传播，可以看到沿时间步反向传递的梯度，在所有的隐藏节点上，总是和同一个矩阵做乘积运算。

设 $\boldsymbol{\Lambda}$ 是 n 阶矩阵 $\boldsymbol{A}$ 的特征值组成的对角阵：

$$\boldsymbol{\Lambda} = \mathrm{diag}(\lambda_1, \lambda_2, \cdots, \lambda_n)$$

$\boldsymbol{P}$ 是 $\boldsymbol{A}$ 的特征向量张成的 n 阶矩阵。

由于：

$$\begin{aligned}
\boldsymbol{A} &= \boldsymbol{P}\boldsymbol{\Lambda}\boldsymbol{P}^{-1} \\
\boldsymbol{A}^2 &= \boldsymbol{P}\boldsymbol{\Lambda}\boldsymbol{P}^{-1}\boldsymbol{P}\boldsymbol{\Lambda}\boldsymbol{P}^{-1} \\
&= \boldsymbol{P}\boldsymbol{\Lambda}^2\boldsymbol{P}^{-1} \\
&\vdots \\
\boldsymbol{A}^t &= \boldsymbol{P}\boldsymbol{\Lambda}^t\boldsymbol{P}^{-1} \qquad \blacksquare
\end{aligned}$$

反向传播时，模型在同一层共享权值参数矩阵，每个时间步上迭代做矩阵乘积运算，梯度的缩放趋势同样受到权值参数矩阵的特征值或奇异值（非方阵）的影响。梯度趋近于 0 和无穷大的情况，分别是第 6 章介绍过的**梯度弥散（vanishing gradient）**和**梯度爆炸（exploding gradient）**。

梯度弥散问题可以通过学习策略的优化方法（RMSprop/Adam）来缓解。后续章节介绍的 RNN 变体模型也可以缓解梯度弥散问题。

梯度爆炸问题，可以权衡牺牲一部分训练效率，通过适当调低学习率参数来改善。

这里介绍另一种应对梯度爆炸问题的方法：**梯度裁剪**。

7.6 梯度裁剪

梯度裁剪（gradient clipping）是一种简单有效的方法。指在梯度突然变大的情况下，可以用特殊手段把梯度收缩（rescale）到特定阈值（threshold）之下。观察梯度变化的方法，是跟踪它的 L_2 范数。当 $||\nabla_{\boldsymbol{W},b}E||_2 \geqslant \text{threshold}$ 时更新梯度值为

$$\nabla_{\boldsymbol{W},b}E = \frac{\text{threshold}}{||\nabla_{\boldsymbol{W},b}E||_2}\nabla_{\boldsymbol{W},b}E \tag{7.18}$$

实践中，一种实现方法是分别取当前时间步下所有权值参数、偏置项的 L_2 范数的平方和再开方，仍然得到一个标量，再和阈值比较大小，来确定梯度的收缩系数，简洁高效地实现计算。新引入的一个超参数 threshold 需要根据数据场景来设置和调整，实验表明（Razvan *et al.*，2012），模型对这个阈值参数不敏感，即使取值很小，裁剪算法也能得到不错结果。

梯度裁剪也可以看成是根据梯度的范数自适应地调整学习率进而调整梯度。而自适应优化方法，是以累计统计量来更新梯度，起到加快模型收敛的作用；梯度裁剪的目标不是加速收敛，而是通过调整每个时间步的梯度来缓解梯度爆炸问题。

7.7 算法实现

基于上述推导，可以**不借助深度学习框架**，实现 RNN 模型的前向和反向传播算法。

前向传播在时间节点和层级两个方向上分别计算：

```
# RNN 沿时间步的前向传播
#输入:
#   - x: 当前时间步的输入数据, shape (N, D).
#   - prev_h: 上一时间步的隐藏节点状态, shape (N, H)
#   - Wx: 输入x到隐藏节点h之间的权值矩阵, shape (D, H)
#   - Wh: 当前和下一个隐藏节点之间的权值矩阵, shape (H, H)
#   - b: 偏置向量Biases, shape (H,)
#返回:
#   - next_h: 下一个时间步的隐藏节点状态, shape (N, H)
#   - cache:是一个元组, 缓存中间变量, 反向传播时复用
def rnn_step_forward(x, prev_h, Wx, Wh, b):
    # 隐藏节点状态原始输出
    z = np.matmul(x,Wx)+np.matmul(prev_h,Wh) +b
    # 原始输出激活作为最终输出
    next_h = np.tanh(z)
    # 计算和缓存反向传播用到的中间变量
    dtanh = 1. - next_h * next_h
    cache=(x, prev_h, Wx, Wh, dtanh)
    return next_h, cache
```

实现了单时间步处理逻辑，就可以在层次和时间两个维度上迭代调用单时

间步处理逻辑，实现多层多时间步的 RNN 前向处理：

```
# RNN 多层组合下沿多个时间步的反向传播
#输入:
#   - x: RNN首层的多时间步输入， shape (N, T, D)
#   - layersNum: RNN的组合层数
#输出:
#   - h: RNN最后一层多时间步输出 shape (N, T, H)
def rnn_forward(self, x):
    h, cache = None, None
    N, T, D = x.shape
    L = self.layersNum
    H = self.rnnParams[0]['b'].shape[0]
    xh = x
    for layer in range(L):
        h = np.zeros((N, T, H))
        h0 = np.zeros((N, H))
        cache = []
        for t in range(T):
            h[:, t, :], tmp_cache = self.rnn_step_forward(
                xh[:, t, :],
            h[:, t - 1, :] if t > 0 else h0,
            self.rnnParams[layer]['Wx'], self.rnnParams[
                layer]['Wh'],
            self.rnnParams[layer]['b'])
            cache.append(tmp_cache)
        xh = h  # 之后以h作为xh用于跨层输入
        self.rnnParams[layer]['h'] = h
        self.rnnParams[layer]['cache'] = cache

    return h  # 返回最后一层作为输出
```

反向传播的单时间步运算同样须在时间和层次两个方向上分别计算当前时间步的误差传递和权值参数梯度：

```
#RNN 沿时间步的反向传播
#输入:
#  -dnext_h: 后一时间步隐藏节点的BP误差,shape(N,H)
```

```
#  -cache: 前向传播缓存的中间变量
#输出:
#  -dx: 反向传播到输入的BP误差,shape (N,D)
#  -dprev_h:后一时间步的隐藏节点回传的BP误差,shape(N,H)
#  -dWx:输入x到隐藏节点h之间权值矩阵的BP梯度,shape(D,H)
#  -dWh:当前和下一隐藏节点间权值矩阵的BP梯度,shape(H,H)
#  -db: 偏置向量bias的梯度, shape(H,)
def rnn_step_backward(dnext_h, cache):
    # 加载前向传播是缓存的中间变量
    x, prev_h, Wx, Wh, dtanh = cache
    # 隐藏节点原始输出环节的BP误差
    dz = dnext_h * dtanh
    # 输入环节的BP误差
    dx = np.matmul(dz,Wx.T)
    # 前一时间步隐藏节点环节的BP误差
    dprev_h = np.matmul(dz,Wh.T)
    # 梯度计算
    dWx = np.matmul(x.T,dz)
    dWh = np.matmul(prev_h.T,dz)
    db = np.sum(dz,axis=0)

    # 外层合并各时间步梯度, 得到最终参数梯度
    return dx, dprev_h, dWx, dWh, db
```

多层多时间步 RNN 的反向传播处理：

```
#多层RNN 沿时间步的反向传播
#输入:
#   - dh: 后一隐藏层返回的BP误差,shape (N, T, H)
#   - x : 首层的原始输入, shape (N, T, D)
#输出:
#   - dx: 反向传播前一层到输入BP误差, shape (N, T, D)
#   - dweight: 是一个参数梯度的列表,
#              列表中的每个元素是一个处理层参数梯度组成的
    元组
def rnn_backward(self, dh):
    N, T, H = dh.shape
    x, _, _, _, _ = self.rnnParams[0]['cache'][0]
```

```
12      D = x.shape[1]
13
14      # 初始化最上一层误差
15      dh_prevl = dh
16      # 保存各层dwh,dwx和db
17      dweights = []
18      # 逐层倒推
19      for layer in range(self.layersNum - 1, -1, -1):
20          # 得到前向传播保存的cache数组
21          cache = self.rnnParams[layer]['cache']
22          DH = D if layer == 0 else H
23          dx = np.zeros((N, T, DH))
24          dWx = np.zeros((DH, H))
25          dWh = np.zeros((H, H))
26          db = np.zeros(H)
27          dprev_h_t = np.zeros((N, H))
28          # 倒序遍历
29          for t in range(T - 1, -1, -1):
30              dx[:, t, :], dprev_h_t, dWx_t, dWh_t, db_t =
                    self.rnn_step_backward(
31              dh_prevl[:, t, :] + dprev_h_t,
32              cache[t])
33              dWx += dWx_t
34              dWh += dWh_t
35              db += db_t
36          # 本层得出的dx, 作为下一层的prev_l
37          dh_prevl = dx
38
39          dweight = (dWx, dWh, db)
40          dweights.append(dweight)
41
42      # 返回x误差和各层参数误差
43      return dx, dweights
```

以上源码，实现了 RNN 模型的前向和反向传播的核心算法。

7.8 目标问题：序列数据分析

可以用一组序列数据作为目标问题，来观察 RNN 模型的运算机制。

7.8.1 数据准备

把正弦函数和余弦函数叠加，用来随机生成训练和验证数据：

$$y = \sin(\frac{\pi x}{3}) + \cos(\frac{\pi x}{3}) + \sin(\frac{3\pi x}{2}) \tag{7.19}$$

产生序列数据的图像如图7.5所示。

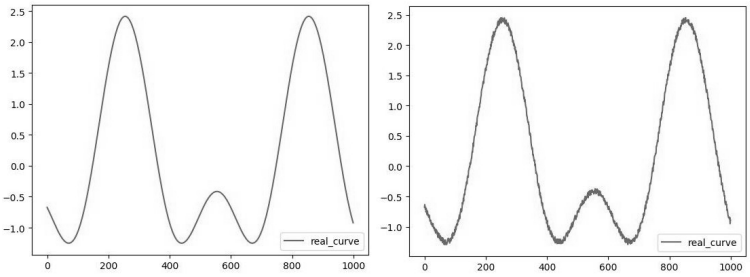

图 7.5 准备序列数据，
左图：原始图像，右图：增加噪声后的图像

图7.5左边的图形是叠加函数的原始图像，右边是给数据增加随机噪声后的图像，用增加噪声后的函数输出，采样生成离散数据集。

用 10 个采样时间步作为一个样本片段，每 32 个样本组成一个 mini-batch，用来训练参数。训练完成后，把包含 10 个采样时间步的独立的验证样本送入模型，推理预测每个样本片段第 11 个时间步的输出。

用下面的步骤定义超参数，生成数据集：

```
import numpy as np
import logging.config
import random, time
import matplotlib.pyplot as plt

import sys
import os
# 定义超参数
class Params:
```

```
EPOCH_NUM = 5  # EPOCH
MINI_BATCH_SIZE = 32  # batch_size
ITERATION = 1  # 每batch训练轮数
#学习率
LEARNING_RATE = 0.005
VAL_FREQ = 50  # val per how many batches
LOG_FREQ = 1000000  # log per how many batches
# RNN中隐藏节点的个数,每个时间节点上的隐藏节点的个数, 是w
    的维度
HIDDEN_SIZE = 30
NUM_LAYERS = 3  # RNN/LSTM的层数
# 设置默认数值类型
DTYPE_DEFAULT = np.float32
INIT_W = 0.01  # 权值初始化参数

TIMESTEPS = 10  # 循环神经网络的训练序列长度
PRED_STEPS = TIMESTEPS  # 预测序列长度
TRAINING_STEPS = 10000  # 训练轮数
TRAINING_EXAMPLES = 10000  # 训练数据个数
TESTING_EXAMPLES = 1000  # 测试数据个数
SAMPLE_GAP = 0.01  # 采样间隔
# 验证集大小
VALIDATION_CAPACITY = TESTING_EXAMPLES-TIMESTEPS

# Optimizer params
BETA1 = 0.9
BETA2 = 0.999
EPS = 1e-8
EPS2 = 1e-10

# 序列数据生成类
class SeqData(object):
    # 数据类用no.float32
    def __init__(self, dataType):
        self.dataType = dataType
        self.x, self.y,self.x_v, self.y_v = self.initData
            ()
```

```
45          # 训练样本范围
46          self.sample_range = [i for i in range(len(self.y))
                ]
47          # 验证样本范围
48          self.sample_range_v = [i for i in range(len(self.
                y_v))]
49
50      # 产生原始训练数据
51      def initData(self):
52          # 用正弦和余弦函数生成训练和测试数据集合
53          # (1w+1k)×0.01
54          test_start = (Params.TRAINING_EXAMPLES + Params.
                TIMESTEPS) * Params.SAMPLE_GAP
55          #  (1w+1k)×0.01 + (1w+1k)×0.01
56          test_end = test_start + (Params.TESTING_EXAMPLES +
                 Params.TIMESTEPS) * Params.SAMPLE_GAP
57
58          # np.linspace 生成等差数列(start-首项、
59          #   stop-尾项、number-项数)
60          # endpoint-末项是否包含在内,
61          #   默认为true, 即包含在内
62          # 从等差数列生成函数值序列
63          train_X, train_y = self.generate_data(self.curve(
                np.linspace(
64          0, test_start, Params.TRAINING_EXAMPLES + Params.
                TIMESTEPS, dtype=self.dataType)))
65          test_X, test_y = self.generate_data(self.curve(np.
                linspace(
66          test_start, test_end, Params.TESTING_EXAMPLES +
                Params.TIMESTEPS, dtype=self.dataType)))
67          # 返回训练数据集和验证数据集
68          return train_X, train_y,test_X,test_y
69
70      #产生数据
71      #1.序列的第i项和后面的TIMESTEPS-1项,
72      # 合在一起共10个元素序列作为输入,
73      # 第i + TIMESTEPS项个元素作为输出
```

```
74      #2.用curve函数生成从i开始的10个元素的信息，
75      # 预测第i + TIMESTEPS顺位的元素值
76      #3.一共生成TRAINING_EXAMPLE=10000对“序列-值”对
77      # 用于训练，同理生成验证数据
78      def generate_data(self,seq):
79          X = []
80          y = []
81
82          # 交换维度为N,T,D :(N,10,1)->(N,1,10)
83          for i in range(len(seq) - Params.TIMESTEPS-Params.
                PRED_STEPS):
84              X.append([seq[i: i + Params.TIMESTEPS]])
85              y.append([seq[i + Params.TIMESTEPS:i + Params.
                    TIMESTEPS + Params.PRED_STEPS]])
86
87          return np.swapaxes(np.array(X, dtype=self.dataType
                ),1,2),
88           np.swapaxes(np.array(y, dtype=self.dataType),1,2)
89
90      # 定义数据生成的规则
91      def curve(self,x):
92          return np.sin(np.pi * x / 3.) + np.cos(np.pi * x /
                 3.)
93           + np.sin(np.pi * x / 1.5)++ np.random.uniform
                (-0.05,0.05,len(x))
94
95      # 对训练样本序号随机分组
96      def getTrainRanges(self, miniBatchSize):
97          rangeAll = self.sample_range
98          random.shuffle(rangeAll)
99          rngs = [rangeAll[i:i + miniBatchSize] for i in
                range(0,
100          len(rangeAll), miniBatchSize)]
101         return rngs
102
103     # 获取训练样本范围对应的图像和标注
104     def getTrainDataByRng(self, rng):
```

```
105         xs = np.array([self.x[sample] for sample in rng],
                self.dataType)
106         values = np.array([self.y[sample] for sample in
                rng])
107         return xs, values
108
109     # 获取验证样本，同一分组内的样本不打乱，用于显示连续曲
            线
110     def getValData(self, valCapacity):
111         samples_v = [i for i in range(valCapacity)]
112         x_v = np.array([self.x_v[sample_v]
113             for sample_v in samples_v], dtype=self.
                    dataType)
114         #  正确类别 1×K
115         y_v = np.array([self.y_v[sample_v] for sample_v in
                 samples_v])
116         return x_v, y_v
```

如果用均方误差（Mean-Square Error，MSE）作为损失函数，D 维样本数据的预测损失为

$$\text{Loss} = \frac{1}{2}\sum_{i=0}^{D-1}(\hat{y}_i - y_i)^2 \tag{7.20}$$

在这个序列预测场景中，以一个时段的离散采样数据序列作为输入，预测下一刻的输出；在 RNN 最后一层，各个时间节点可以有多个输出，在这个场景下，只需获取最后一个时间节点的数值结果 $y = y_t$，即 N 步输入，1 步输出的 N **v 1** 方式来计算预测损失。

误差损失为

$$\frac{\partial \text{Loss}}{\partial y_t} = \frac{\partial}{\partial y_t}\frac{1}{2}(\hat{y}_t - y_t)^2 = \frac{1}{2}\frac{\partial[(\hat{y}_t - y_t)^2]}{\partial(\hat{y}_t - y_t)}\frac{\partial(\hat{y}_t - y_t)}{\partial y_t} = y_t - \hat{y}_t \tag{7.21}$$

将这个误差损失反向传递给模型，即可使用 BPTT 算法得到各个层级节点的误差和权值参数梯度。

```
1 # MSE误差计算类
2 class MseLoss:
```

```
@staticmethod
def loss(y,y_, n):
    corect_logprobs = Tools.mse(y, y_)
    data_loss = np.sum(corect_logprobs) / n
    delta = (y - y_) / n

    return data_loss, delta ,None
```

7.8.2 模型搭建

用上面实现 Vanilla RNN 的算法搭建一个三层 RNN 模型，如图7.6所示。

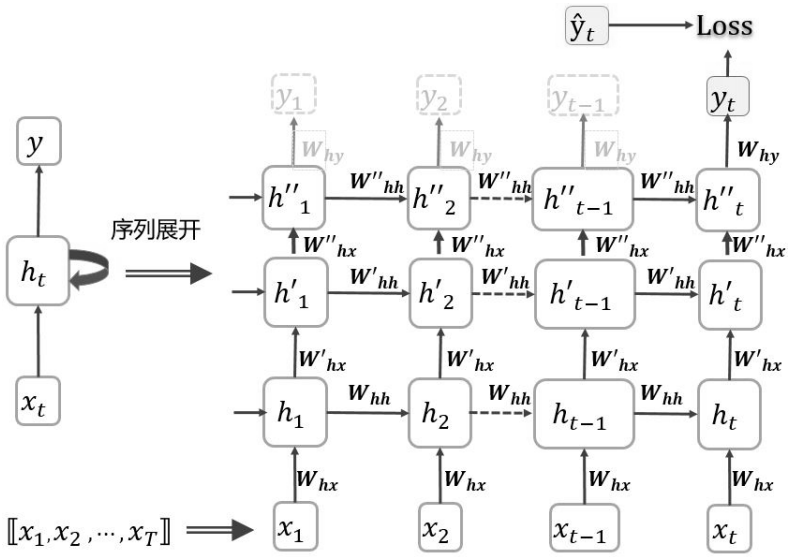

图 7.6　三层 RNN 模型结构：N v 1

最后的 y_t 节点作为模型预测输出，和真实结果 $\hat{y}_t$ 一起计算误差损失。

这个模型的输入数据为 32×10，每层 RNN 有 10 个时间步。

模型每层的结构和参数量情况：

- 第一层，每个时间步的隐藏节点维度为 30，在时间步方向上的参数量为 $30 \times 30 + 30 = 930$，在层次方向上的参数量为 $1 \times 30 + 30 = 60$，第一层合计 990 个参数。

- 第二层结构在时间步方向的结构和第一层相同，参数也有 930 个，在层次方向上，由于输入数据的维度为 30，所以参数量也是 $30 \times 30 + 30 = 930$，第二层合计 1860 个参数。
- 第三层在时间步和层两个方向的结构均和第二层相同，参数总量也是 1860 个，但是在目标问题场景下，只需要取最后一个时间步的输出。
- 最后一层全连接层，输入维度 32×30，输出维度 32×1，全连接层共有 $30 \times 1 + 1 = 31$ 个参数。

在这个三层 RNN 结构里，每一层都有 10 个时间步，虽然每个时间步都有 30 个隐藏节点分别和下一时间步、前一层同时间步节点做结构上的全连接，然而由于每一层相同方向上的训练参数是共享的，所以整个模型的总参数量并不算多。各处理层参数量情况如表7.1所示。

表 7.1 三层 RNN 模型的参数量

模型层	序列方向	层次方向	参数总量
输入	无	\	\
RNN-L1	930	60	990
RNN-L2	930	930	1860
RNN-L3	930	930	1860
FC	\	30×1+1=31	31

RNN 第一层和之后各层，由于层次方向上输入数据维度不同，参数量也不同。

这个预测场景的学习策略优化算法同样**不借助深度学习框架**，复用之前章节中实现的 **Adagrad 算法**。

7.8.3 验证结果

观察图7.7，经过 600 个 mini-batch 的迭代训练，模型在训练和验证数据上都可以正常收敛。

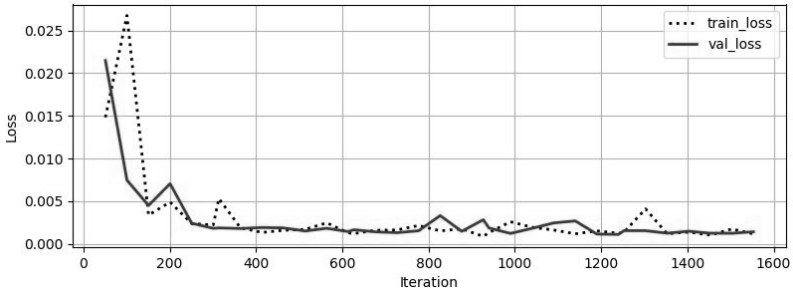

图 7.7　RNN 模型训练的训练损失曲线

跟踪训练和验证的情况可以观察到，4 轮训练之后，模型在验证数据集上的损失趋于稳定：

```
start..
w,b init..
epoch  0, loss=0.026765, loss_v=0.007466
epoch  1, loss=0.001339, loss_v=0.001915
epoch  2, loss=0.001207, loss_v=0.001658
epoch  3, loss=0.001321, loss_v=0.001861
epoch  4, loss=0.001743, loss_v=0.001246
end
```

推理预测结果如图7.8所示, 预测输出曲线（predictions）几乎与数据集真实的曲线（real curve）重合。

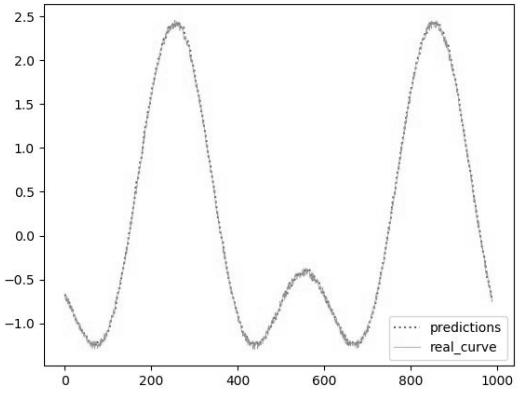

图 7.8　推理结果符合真实数据

7.9 小结

这个实现，由于**不借助深度学习框架**，可以很方便地换用不同的激活函数和学习策略优化方法。也可以根据不同的数据特性，自由定制组合模型构件，灵活适应不同的场景。

循环神经网络引入记忆机制，得以抽取序列特征，且能通过增加隐藏节点和隐藏层级，提高模型的表达能力，开辟了深度学习算法适用场景的新方向。然而，在时序数据中，远近不同样本对输出决策的影响权重往往不同；此外，随着 RNN 模型深度的增加，梯度传递问题也使模型不易训练。

下一章将讨论 Vanilla RNN 模型的变体：**长短时记忆网络 LSTM** 及其对上述问题的解决方法。

拓展阅读

Andrej Karpathy 对循环神经网络反直觉的有效性所做的思考。
The Unreasonable Effectiveness of Recurrent NeuralNetworks.
`http://karpathy.github.io/2015/05/21/rnn-effectiveness`

Andrej Karpathy 在另一篇文章里分享了给 Google 深度学习框架提交 Issue 的经历，并介绍了投入时间去理解反向传播算法的动机。
Yes you should understand backprop.
`https://medium.com/@karpathy/yes-you-should-understand-backprop`

参 考 文 献

[1] J J Hopfield, **Neural networks and physical systems with emergent collective computational abilities.** Proceedings of the National Academy of Sciences of the USA, 1982, 79(8):2554–2558.

[2] Bengio, Yoshua, Simard, Patrice, and Frasconi, Paolo. **Learning long-term dependencies with gradient descent is difficult**. Neural Networks, IEEE Transactions on, 1994, 5(2):157–166.

[3] Razvan Pascanu, Tomas Mikolov, Yoshua Bengio. **Understanding the exploding gradient problem**. arXiv:1211.5063v1, 2012.

[4] Eugene Vorontsov, Chiheb Trabelsi, Samuel Kadoury, Chris Pal. **On orthogonality and learning recurrent networks with long term dependencies**. PMLR, 2017, 70:3570-3578.

第 8 章

长短时记忆网络（LSTM）——指数分析

> “Someday, ··· we'll ··· produce a river of mathematical symbolism that will chart past and future history.”
>
> “有一天，····· 我们将 ····· 创造一条数理之河，标示过去、未来的历史。”
>
> Hari Seldon @ Streeling University.
> Isaac Asimov, Foundation series.

第 7 章介绍并实现循环神经网络，引入记忆机制来抽取序列数据特征，使用模型完成了复合函数的曲线拟合预测。

然而，序列数据中远近不同样本对输出决策的影响权重往往不同；此外，随着 RNN 模型深度增加，梯度传递问题也使模型不易训练。

这一章描述 Vanilla RNN 模型的变体：**长短时记忆网络 LSTM**（Long Short-Time Memory）对上述问题的缓解方法，包括模型、结构和算法，同样

不借助深度学习框架，实现 LSTM 的前向和反向传播算法，再将其应用于沪深 300 指数分析，以验证这个模型。

8.1 目标问题：投资市场的指数分析

提示：本案例仅用于算法研究，**不作为预测 ETF 产品净值走势的参考**。

本章将要介绍的模型选用沪深 300 指数分析作为目标问题。沪深 300 指数反映 A 股市场整体走势，体现证券市场的概貌和运行状况，为指数化投资和指数衍生产品创新提供基础条件。沪深 300 指数具有良好的市场代表性，因而被作为业内投资业绩的基准评价要素之一。

接下来，我们以沪深 300 指数过去有效交易日的成交数据为分析对象，用深度学习算法捕捉与指数变化相关的特征。

8.2 挑战：梯度弥散问题

在 RNN 的 BPTT 算法中，沿时间步迭代反向传播的误差可能导致梯度的爆炸或弥散问题。过大的梯度会造成训练参数在目标区域附近反复摆动，影响模型收敛，可以通过梯度裁剪来部分缓解；而梯度弥散问题，在长时间跨度的 RNN 模型里，会使参数不再更新，训练因而失效。梯度弥散问题，在 LSTM 模型出现之前尚无被广泛采用的有效缓解方法。

LSTM 算法提出一种新的循环网络变体结构，创造性地在模型中引入逻辑门单元，实现“记忆”和“遗忘”机制，以此来解决长程序列在模型中的梯度传递问题。

8.3 长短时记忆网络的结构

LSTM 同 Vanilla RNN 一样，在每个时间步，接收该时间步的输入 $\boldsymbol{x}_t$，以及上一步的隐藏状态 $\boldsymbol{h}_{t-1}$；不同的是，LSTM 也同时保存了一个单元状态 $\boldsymbol{c}_t$，所以每个时间步还会接收上一个时间步的单元状态 $\boldsymbol{c}_{t-1}$。

在每个时间步上，先做加权变换：

$$\boldsymbol{z} = \boldsymbol{W}_x \boldsymbol{x}_t + \boldsymbol{W}_h \boldsymbol{h}_{t-1} + \boldsymbol{b}$$

然后把变换结果 $\boldsymbol{z}$ 拆分为四段 $\boldsymbol{z}_i, \boldsymbol{z}_f, \boldsymbol{z}_o, \boldsymbol{z}_g$，分别作为输入门、遗忘门、

输出门和准单元状态：

$$
\begin{aligned}
&\text{输入门：} && \boldsymbol{i} = \sigma(\boldsymbol{z}_i) \\
&\text{遗忘门：} && \boldsymbol{f} = \sigma(\boldsymbol{z}_f) \\
&\text{输出门：} && \boldsymbol{o} = \sigma(\boldsymbol{z}_o) \\
&\text{准单元状态：} && \boldsymbol{g} = \tanh(\boldsymbol{z}_g)
\end{aligned}
$$

上式中的 σ 函数为 sigmoid 激活函数，tanh() 是 vanilla RNN 中使用过的双曲正切函数，$\boldsymbol{g} = \tanh(\boldsymbol{z}_g)$ 有时被称为单元状态，本书中把 $\boldsymbol{g} = \tanh(\boldsymbol{z}_g)$ 称为**准单元状态**，是为了与当前时间步最后输出的单元状态 $\boldsymbol{c}_t$ 加以区分。

最后，计算下一时间步需要用到的单元状态 $\boldsymbol{c}_t$ 和隐状态 $\boldsymbol{h}_{t-1}$：

$$
\boldsymbol{c}_t = \boldsymbol{f} \otimes \boldsymbol{c}_{t-1} + \boldsymbol{i} \otimes \boldsymbol{g} \qquad\qquad \boldsymbol{h}_t = \boldsymbol{o} \otimes \tanh(\boldsymbol{c}_t)
$$

LSTM 在一个时间步上的计算过程如图8.1所示。

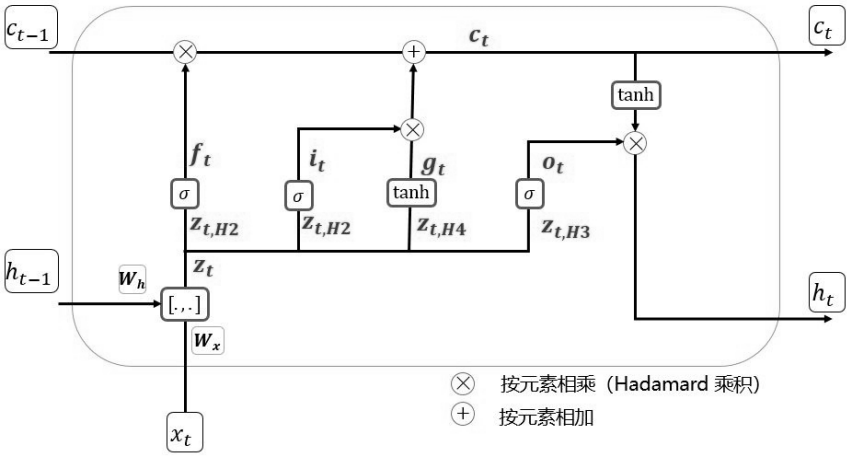

图 8.1 LSTM 单个时间步的模型结构

模型结构中，$\boldsymbol{W}_h$，$\boldsymbol{W}_x$ 及偏置 $\boldsymbol{b}$ 是训练得到的参数。

激活函数 σ 把 $(-\infty, +\infty)$ 之内的实数值变换到区间 [0,1]，数据经过遗忘门 $\boldsymbol{f}$ 的 σ 函数激活后，再与上一时间步输出的单元状态 $\boldsymbol{c}_{t-1}$ 做 Hamadard 按位相乘运算，使输出接近 0 的位置，内容被遗忘；接近 1 位置的部分，内容被保留。

输入门 $\boldsymbol{i}$ 和准单元状态 $\boldsymbol{g}$ 一起构成记忆门，输出需要记住的新特征。

遗忘和记忆的结果逐元素相加，成为新的单元状态 $\boldsymbol{c}_t$ 。

输出门 $\boldsymbol{o}$ 用于计算下一个隐状态 $\boldsymbol{h}_t$，所得每个元素数值，同样经过 σ 函数激活，最后的输出都在 [0,1] 之间。输出门计算得到的下一个隐状态 $\boldsymbol{h}_t$ 接下来会与下一个序列的输入拼接在一起，共同进行下一个时间步各个逻辑门的运算。

图8.1上部的从 $\boldsymbol{c}_{t-1}$ 到 $\boldsymbol{c}_t$ 这条传递单元状态的通道，缓解了长程时间序列中，反向传播的梯度弥散问题。将不同时间步贯穿起来，像是传递单元状态特征的“绿色通道”，使 LSTM 模型能在长达 1000 个时间步的含噪序列上（Hochreiter *et al.*，1997），仍能有效地训练和收敛。

接下来，把上述计算过程描述为更一般的 LSTM 模型前向传播算法表达式，以方便反向传播推导。

8.4 LSTM 前向传播算法

先对输入向量 $\boldsymbol{x}_t$ 和上一时间步的隐状态 $\boldsymbol{h}_{t-1}$ 做一次加权变换：

$$\boldsymbol{z}_t = \boldsymbol{W}_x\boldsymbol{x}_t + \boldsymbol{W}_h\boldsymbol{h}_{t-1} + \boldsymbol{b} \tag{8.1}$$

把变换结果拆分为四部分，拆分的方法是把 $\boldsymbol{z}_t$ 在与隐藏节点数对应的 H 这个维度，等分为四份，分别激活：

$$\boldsymbol{i}_t = \text{sigmoid}(\boldsymbol{z}_{t,H1}) \quad \text{（输入门）} \tag{8.2}$$

$$\boldsymbol{f}_t = \text{sigmoid}(\boldsymbol{z}_{t,H2}) \quad \text{（遗忘门）} \tag{8.3}$$

$$\boldsymbol{o}_t = \text{sigmoid}(\boldsymbol{z}_{t,H3}) \quad \text{（输出门）} \tag{8.4}$$

$$\boldsymbol{g}_t = \tanh(\boldsymbol{z}_{t,H4}) \quad \text{（准单元状态）} \tag{8.5}$$

再计算得当前时间步的隐状态输出：

$$\boldsymbol{c}_t = \boldsymbol{f}_t \otimes \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \otimes \boldsymbol{g}_t \tag{8.6}$$

以及当前时间步的单元状态输出：

$$\boldsymbol{h}_t = \boldsymbol{o}_t \otimes \tanh(\boldsymbol{c}_t) \quad \blacksquare \tag{8.7}$$

隐状态输出 $\boldsymbol{h}_t$ 与单元状态输出 $\boldsymbol{c}_t$，连同前一层下一时间步的隐状态输出 $\boldsymbol{h}_{t+1}^{l-1}$ 一起，作为 LSTM 当前层下一个时间步的输入。

LSTM 的算法表达式，同 Vanilla RNN 类似，也可以把 $\boldsymbol{W}_x$ 和 $\boldsymbol{W}_h$ 拼接为一个 $\boldsymbol{W}$ 权值参数矩阵，同时把前一层输入 $\boldsymbol{x}_t$ 和前一时刻隐藏节点输出 $\boldsymbol{h}_{t-1}$ 拼接成为 $[\boldsymbol{x}_t, \boldsymbol{h}_{t-1}]$，再和权参 $\boldsymbol{W}$ 做仿射变换：

$$\boldsymbol{z}_t = \boldsymbol{W}[\boldsymbol{x}_t, \boldsymbol{h}_{t-1}] + \boldsymbol{b} \tag{8.8}$$

训练和预测时，只要表达式保持一致，两种方法的结果是等价的。这里采用权参分开来书写表达式，同样是为了便于观察反向传播算法的推导步骤。

8.5 LSTM 反向传播算法

观察图8.2所示的多层多时间步的 LSTM 模型。

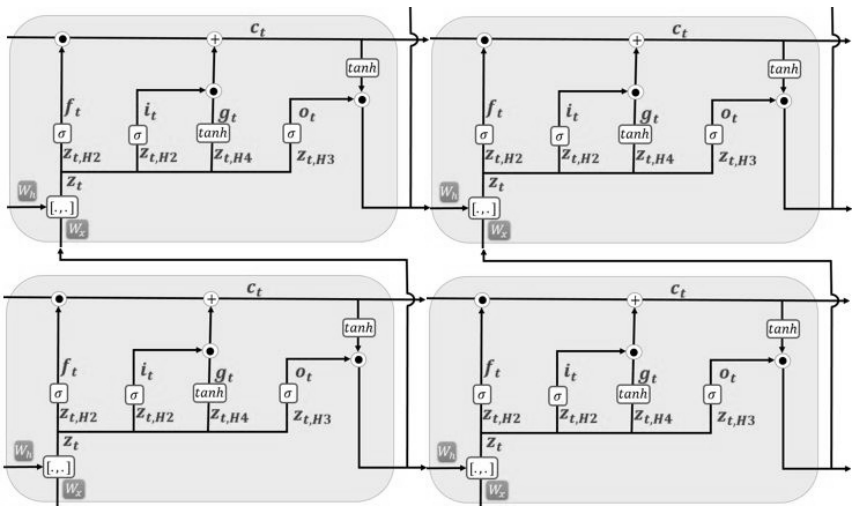

图 8.2 多层多时间步 LSTM 结构

训练得到的误差损失，从时间步和层次两个方向进行反向传播计算。

8.5.1 误差反向传播

第 l 层 t 时刻的误差损失，来自后一时间步和前一层同时间步的误差叠加：

$$\delta \boldsymbol{h}_t = \frac{\partial E}{\partial \boldsymbol{h}_t^l} = \frac{\partial E_t^{l+1}}{\partial \boldsymbol{h}_t^l} + \frac{\partial E_{t+1}^l}{\partial \boldsymbol{h}_t^l} = \delta \boldsymbol{h}_t^{l+1} + \delta \boldsymbol{h}_{t+1}^l \tag{8.9}$$

推得输出门 $\boldsymbol{o}_t$ 和单元状态 $\boldsymbol{c}_t$ 的反向传递误差：

$$\delta \boldsymbol{o}_t = \frac{\partial E}{\partial \boldsymbol{h}_t} \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{o}_t} = \delta \boldsymbol{h}_t \otimes \tanh'(\boldsymbol{c}_t) \tag{8.10}$$

单元状态 $\boldsymbol{c}_t$ 总的回传误差，还需要加上 $\boldsymbol{c}_{t+1}$ 通道回传的误差部分：

$$\delta \boldsymbol{c}_t = \frac{\partial E}{\partial \boldsymbol{h}_t} \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{c}_t} + \delta \boldsymbol{c}_{t+1} = \delta \boldsymbol{h}_t \otimes \boldsymbol{o}_t \otimes \tanh'(\boldsymbol{c}_t) + \delta \boldsymbol{c}_{t+1} \tag{8.11}$$

误差继续回传：

$$\delta \boldsymbol{i}_t = \frac{\partial E}{\partial \boldsymbol{c}_t} \frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{i}_t} = \delta \boldsymbol{c}_t \otimes \boldsymbol{g}_t \tag{8.12}$$

$$\delta \boldsymbol{f}_t = \frac{\partial E}{\partial \boldsymbol{c}_t} \frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{f}_t} = \delta \boldsymbol{c}_t \otimes \boldsymbol{c}_{t-1} \tag{8.13}$$

$$\delta \boldsymbol{g}_t = \frac{\partial E}{\partial \boldsymbol{c}_t} \frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{g}_t} = \delta \boldsymbol{c}_t \otimes \boldsymbol{i}_t \tag{8.14}$$

$$\delta \boldsymbol{c}_{t-1} = \frac{\partial E}{\partial \boldsymbol{c}_t} \frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{c}_{t-1}} = \delta \boldsymbol{c}_t \otimes \boldsymbol{f}_t \tag{8.15}$$

继续推出激活前，加权变换输出环节误差的四个分量:

$$\delta \boldsymbol{z}_{H1,t} = \delta \boldsymbol{i}_t \otimes \text{sigmoid}'(\boldsymbol{z}_{H1,t}) \tag{8.16}$$

$$\delta \boldsymbol{z}_{H2,t} = \delta \boldsymbol{f}_t \otimes \text{sigmoid}'(\boldsymbol{z}_{H2,t}) \tag{8.17}$$

$$\delta \boldsymbol{z}_{H3,t} = \delta \boldsymbol{o}_t \otimes \text{sigmoid}'(\boldsymbol{z}_{H3,t}) \tag{8.18}$$

$$\delta \boldsymbol{z}_{H4,t} = \delta \boldsymbol{g}_t \otimes \tanh'(\boldsymbol{z}_{H4,t}) \tag{8.19}$$

8.5.2 激活函数的导函数和参数梯度

观察式 (8.19)，加权变换输出误差的第四个分量 $\delta \boldsymbol{z}_{H4,t}$ 的表达式包含双曲正切函数 tanh 的导函数，可参考上一章对改导函数的推导与特点描述。

对前三个分量表达式中的 sigmoid 函数：

$$\sigma(z) = y = \frac{1}{1+\mathrm{e}^{-z}} \tag{8.20}$$

其导函数为

$$\frac{\mathrm{d}\sigma(z)}{\mathrm{d}z} = -\frac{\mathrm{d}(1+\mathrm{e}^{-z})}{\mathrm{d}z}\frac{1}{(1+\mathrm{e}^{-z})^2} \tag{8.21}$$

$$= -\frac{-\mathrm{e}^{-z}}{(1+\mathrm{e}^{-z})^2} = \frac{1}{1+\mathrm{e}^{-z}}\frac{\mathrm{e}^{-z}}{1+\mathrm{e}^{-z}} \tag{8.22}$$

$$= \frac{1}{1+\mathrm{e}^{-z}}(1-\frac{1}{1+\mathrm{e}^{-z}}) \tag{8.23}$$

$$= \sigma(z)(1-\sigma(z)) \tag{8.24}$$

所以

$$\begin{aligned} \sigma(z) &= y \\ \sigma'(z) &= y(1-y) \end{aligned} \tag{8.25}$$

这说明，前向传播时，保留 S 型函数的激活结果，同样可以在反向传播时直接复用，从而节省算力消耗。

合并的四个分量误差的反向传递结果，成为完整误差矩阵：

$$\delta \boldsymbol{z}_t = [\delta \boldsymbol{z}_{H1,t}, \delta \boldsymbol{z}_{H2,t}, \delta \boldsymbol{z}_{H3,t}, \delta \boldsymbol{z}_{H4,t}] \tag{8.26}$$

继续反向传递：

$$\delta \boldsymbol{x}_{t-1} = \boldsymbol{W}_{\boldsymbol{x}}^{\mathrm{T}} \delta \boldsymbol{z}_t \tag{8.27}$$

$$\delta \boldsymbol{h}_{t-1} = \boldsymbol{W}_{\boldsymbol{h}}^{\mathrm{T}} \delta \boldsymbol{z}_t \tag{8.28}$$

可以推得各时间步反向传播的参数梯度：

$$\nabla_{\boldsymbol{W}_{x,t}} E = \delta \boldsymbol{z}_t (\boldsymbol{x}_t)^{\mathrm{T}} \tag{8.29}$$

$$\nabla_{\boldsymbol{W}_{f,t}} E = \delta \boldsymbol{z}_t (\boldsymbol{h}_{t-1})^{\mathrm{T}} \tag{8.30}$$

$$\nabla_{\boldsymbol{b},t} E = \boldsymbol{z}_t \tag{8.31}$$

最后加总 LSTM 当前一层各个时间步上的参数梯度，得到当前层各训练参数最终的更新梯度：

$$\nabla_{\mathrm{Param}} E = \sum_{t=0}^{T} \nabla_{\mathrm{Param}_t} E \qquad \blacksquare \tag{8.32}$$

以上，完成了 LSTM 核心模型在多层多时间步上的全部误差损失反向传播和参数梯度计算的推导。

8.6 算法实现

根据 LSTM 的前向和反向传播算法，可以不借助深度学习框架，实现这个模型。

8.6.1 实现 LSTM 单时间步的前向计算

每个时间步上的前向计算，使用本层输入和上一时间步的隐藏状态，计算当前时间步的隐藏状态输出：

```
# LSTM 沿单个时间步的前向传播
#输入:
#    - x: 当前时间步的输入数据, shape（N, D）
#    - prev_h: 上一时间步的隐藏节点h状态, shape（N, H）
#    - prev_c: 上一时间步的隐藏节点c状态, shape（N, H）
#    - Wx: 输入x到隐藏节点之间的权值矩阵, shape（D, 4H）
#    - Wh: 隐藏节点之间的权值矩阵, shape（H, 4H）
#    - b: 偏置向量Biases, shape（4H,）
#返回:
#    - next_h: 下一个时间步的隐藏节点h状态, shape（N, H）
#    - next_c: 下一个时间步的隐藏节点c状态, shape（N, H）
```

```
#   - cache:是一个元组（Tuple），缓存中间变量，反向传播时复用
def lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b):
    # 首层x（N,T,D），向上传递后变成xh（N,T,H）
    # 首层 Wx（D,H），  向上传递后变成Wxh（H,H）
    H = prev_h.shape[1]
    z = Tools.matmul(x, Wx) + Tools.matmul(prev_h, Wh) + b
    # 输入门
    i = Tools.sigmoid(z[:, :H])
    # 遗忘门
    f = Tools.sigmoid(z[:, H:2 * H])
    # 输出门
    o = Tools.sigmoid(z[:, 2 * H:3 * H])
    # 准单元状态
    g = np.tanh(z[:, 3 * H:])
    next_c = f * prev_c + i * g
    next_h = o * np.tanh(next_c)
    # 缓存中间变量，用于反向传播
    cache = (x, prev_h, prev_c, Wx, Wh, i, f, o, g, next_c)
    return next_h, next_c, cache
```

8.6.2 实现 LSTM 多层多时间步的前向计算

多层叠加的 LSTM 需要在每一层做多时间步下的前向计算，得到当前层全部隐藏节点向后一层的输出，再使用同样的方法逐层迭代计算，得到最后一层的输出。取用最后一层的全部输出，即 $N \mathrm{v} N$ 模型；若最后一层只取最后一个时间步的输出，即是 $N \mathrm{v} 1$ 模型。

```
# LSTM 多层组合下沿多个时间步的反向传播
#输入:
#   - x: LSTM首层的多时间步输入，  shape (N, T, D)
#   - layersNum: LSTM的组合层数
#输出:
#   - h: LSTM最后一层多时间步输出, shape (N, T, H)
def lstm_forward(self,x):
    # 获取x各维度规格
```

```
    N, T, D = x.shape
    # 多层LSTM结构预定义的层数
    L = self.layersNum
    # 计算输出维度，除法运算后，对浮点数结果取整
    H = int(self.lstmParams[0]['b'].shape[0] / 4)
    # 首层输入是x
    xh = x
    #多层叠加，逐层计算
    for layer in range(L):
        # 对每一层初始化中间变量
        h = np.zeros((N, T, H))
        h0 = np.zeros((N, H))
        c = np.zeros((N, T, H))
        c0 = np.zeros((N, H))
        cache = []
        # 当前层的多时间步，逐步计算
        for t in range(T):
            h[:, t, :], c[:, t, :],tmp_cache =
                lstm_step_forward(
                xh[:, t, :],
                h[:, t - 1, :] if t > 0 else h0,
                c[:, t - 1, :] if t > 0 else c0,
                lstmParams[layer]['Wx'],
                lstmParams[layer]['Wh'],
                lstmParams[layer]['b'])
            cache.append(tmp_cache)
    # 从第二层开始，以xh作为跨层输入
    xh = h
    # 缓存当前层的中间结果，反向传播时使用
    lstmParams[layer]['h'] = h
    lstmParams[layer]['c'] = c
    # 保存当前层各时间步的中间结果
    lstmParams[layer]['cache'] = cache
    return h
```

8.6.3 实现 LSTM 单时间步的反向传播

单个时间步的反向传播，复用前向计算时保存的中间变量，降低运算量：

```
#LSTM 沿单个时间步的反向传播
#输入:
#    - dnext_h: 后一时间步隐藏节点h的BP误差,shape（N, H）
#    - dnext_c: 后一时间步隐藏节点c的BP误差,shape（N, H）
#    - cache: 前向传播缓存的中间变量
#输出:
#    - dx: 反向传播到输入的BP误差, shape（N, D）
#    - dprev_h: 后一时间步的隐藏节点h回传的BP误差, shape（N, H）
#    - dprev_c: 后一时间步的隐藏节点c回传的BP误差, shape（N, H）
#    - dWx: 输入x到隐藏节点h之间权值矩阵的BP梯度, shape（D, 4H）
#    - dWh: 当前和下一个隐藏节点之间权值矩阵的BP梯度,shape（H,4H）
#    - db: 偏置向量bias的梯度, shape（4H,）
def lstm_step_backward(dnext_h, dnext_c, cache):
    # 加载前向传播时保留的中间结果
    x, prev_h, prev_c, Wx, Wh, i, f, o, g, next_c = cache
    # 在后一隐状态c环节的BP误差
    dnext_c = dnext_c + o * (1 - np.tanh(next_c) ** 2) *
        dnext_h
    # 各个逻辑门的BP误差
    di = dnext_c * g
    df = dnext_c * prev_c
    do = dnext_h * np.tanh(next_c)
    dg = dnext_c * i
    # 在当前隐状态c环节的BP误差
    dprev_c = f * dnext_c
    # 完整误差损失由共四部分合并得到
    dz = np.hstack((i * (1 - i) * di,
        f * (1 - f) * df,
        o * (1 - o) * do,
        (1 - g ** 2) * dg))
```

```
    # 输入环节的误差损失
    dx = Tools.matmul(dz, Wx.T)
    # 当前隐状态h环节的BP误差
    dprev_h = Tools.matmul(dz, Wh.T)
    # 当前时间步的参数梯度
    dWx = Tools.matmul(x.T, dz)
    dWh = Tools.matmul(prev_h.T, dz)
    db = np.sum(dz, axis=0)
    # 返回当前时间步的误差损失和参数梯度
    return dx, dprev_h, dprev_c, dWx, dWh, db
```

8.6.4 实现 LSTM 多层多时间步的反向传播

多层叠加 LSTM 在多时间步下的反向传播，自后向前，对每一层、每个时间步逆向运算：

```
#多层LSTM 沿时间步的反向传播
#输入:
#   - dh: 后一隐藏层返回的BP误差,shape（N, T, H)
#   - x : 首层的原始输入, shape（N, T, D)
#输出:
#   - dx: 反向传播前一层到输入BP误差, shape（N, T, D)
#   - dweight: 是一个参数梯度的列表,
#              列表中的每个元素是一个处理层参数梯度组成的元组
def lstm_backward(self,dh,x)
    # BP误差的各个维度规格
    N, T, H = dh.shape
    D = x.shape[1]
    dh_prevl = dh
    # 初始化集合变量, 保存各层dwh,dwx和db
    dweights = []
    # 逐层反向传播
    for layer in range(self.layersNum - 1, -1, -1):
        # 得到前向传播保存的cache数组
        cache = self.lstmParams[layer]['cache']
        # 反向传播到第一层, 恢复原始输入的规格
```

```
21          DH = D if layer == 0 else H
22          # 每一层，初始化BP误差和梯度
23          dx = np.zeros((N, T, DH))
24          dWx = np.zeros((DH, 4 * H))
25          dWh = np.zeros((H, 4 * H))
26          db = np.zeros((4 * H))
27          dprev_h = np.zeros((N, H))
28          dprev_c = np.zeros((N, H))
29          # 每个时间步上反向传播运算
30          for t in range(T - 1, -1, -1):
31              dx[:, t, :], dprev_h, dprev_c, dWx_t, dWh_t,
                    db_t =lstm_step_backward(
32              dh_prevl[:, t, :] + dprev_h,dprev_c,cache[t])
33              # 各时间步的梯度合并得到当前层共享参数的梯度
34              dWx += dWx_t
35              dWh += dWh_t
36              db += db_t
37          # 本层得出的dx，作为下一层的prev_l
38          dh_prevl = dx
39          dweight = (dWx, dWh, db)
40          dweights.append(dweight)
41
42      # 返回x误差和各层参数误差
43      return dx, dweights
```

以上源码实现了 LSTM 模型前向传播和反向传播的核心算法，输入序列数据的时间步长度确定后，不同 LSTM 层保持一致的序列步数，每个时间步隐藏节点的数量，以及 LSTM 叠加的层数，可在构建模型时由入参确定。

8.7 实现沪深 300 指数分析

基于以上实现的 LSTM 的前向和反向传播算法，可以**不借助深度学习框架**，构造长程 RNN 模型，用于沪深 300 指数的时序数据分析。

8.7.1 数据准备

从 2005 年 1 月第一个交易日开始，取 3457 个有效连续交易日的交易数据，作为目标数据集，以收盘指数、成交量和成交金额作为输入数据的三个特征维度。

```
1 , 982.794 , 74.1287 , 44.3198
2 , 992.564 , 71.1911 , 45.2921
3 , 983.174 , 62.8803 , 39.2102
4 , 983.958 , 72.9869 , 47.3747
5 , 993.879 , 57.917 , 37.6293
6 , 997.135 , 58.4908 , 37.0408
7 , 996.748 , 50.1453, 30.933
8 , 996.877 , 60.4407 , 38.4217
9 , 988.306 , 72.9784 , 41.6292
10 , 967.452 , 72.8819 , 42.4981
...
```

用数据的统计特征做规范化因子，对数据做规范化处理，把全部交易日数据按照 64 个交易日为单位进行分组，随机打乱各个分组之间的次序，拆分为独立的训练和验证数据两个集合, 对每个分组使用 64 个交易日的价格指数序列，推理其后 20 个交易日的趋势曲线。

在这个分析样例的模型搭建之前，先集中定义一部分超参数，接下来构造模型时，直接应用这些超参数。

```
import numpy as np
import logging.config
import random, time
import matplotlib.pyplot as plt

import sys
import os
curPath = os.path.abspath(os.path.dirname(__file__))
rootPath = os.path.split(curPath)[0]

# Data
hs300_file = os.path.join("your_data_path/hs300_data_seq.
```

```
    csv")

# General params
class Params:

    EPOCH_NUM = 20  # EPOCH
    MINI_BATCH_SIZE = 32  # batch_size
    # 每batch训练轮数
    ITERATION = 1
    LEARNING_RATE = 0.0015
    VAL_FREQ = 100  # val per how many batches
    LOG_FREQ = 10  # log per how many batches
    # 保留率
    DROPOUT_R_RATE = 0.5
    # 定义LSTM中隐藏节点的个数
    # 每个时间节点上的隐藏节点的个数, 是w的维度
    HIDDEN_SIZE = 64
    # LSTM每个层次的时间节点个数,
    #     由输入数据的元素个数确定
    NUM_LAYERS = 3  # RNN/LSTM的层数
    # 设置默认数值类型
    DTYPE_DEFAULT = np.float32
    # 权值初始化参数
    INIT_W = 0.01
    # 循环神经网络的训练序列长度
    TIMESTEPS = 64
    # 预测序列长度
    PRED_STEPS = 20

    # Optimizer params
    BETA1 = 0.9
    BETA2 = 0.999
    EPS = 1e-8
    EPS2 = 1e-10

# 数据的加载和预处理
class Hs300Data(object):
```

```
49
50      def __init__(self, absPath, dataType):
51          self.dataType = dataType
52          self.data = absPath
53          self.mu=0
54          self.var=0
55          self.x, self.y, self.pred_x = self.initData()
56          # 训练样本范围
57          self.sample_range = [i for i in range(len(self.y))
                ]
58
59      # 加载hs300数据
60      def _load_eft_data(self):
61          seq = np.loadtxt(open(self.data, "rb"),
62           dtype=float, delimiter=",",
63           usecols=(1,2,3), skiprows=0)
64
65          # 对数据做规范化处理
66          self.mu = np.mean(seq, axis=0)
67          xmu = seq - self.mu
68          self.var = np.mean(xmu ** 2, axis=0)
69          seq = (seq - self.mu)/self.var
70          # 返回经过预处理的指数序列
71          return seq
72
73      # 初始化训练和测试数据
74      def initData(self):
75          train_X, train_y, pred_X = self.generate_data(self
                ._load_eft_data())
76          return train_X, train_y, pred_X
77
78      #对训练和预测数据做分组
79      def generate_data(self,seq):
80          X = []
81          y = []
82          for i in range(len(seq) - Params.TIMESTEPS-Params.
                PRED_STEPS):
```

```
83              X.append(seq[i: i + Params.TIMESTEPS])
84              y.append(seq[i + Params.TIMESTEPS:i
85              + Params.TIMESTEPS + Params.PRED_STEPS])
86
87          pred_X=[]
88          pred_X.append(seq[len(seq)-Params.TIMESTEPS:])
89          return np.array(X, dtype=self.dataType), np.array(
                y, dtype=self.dataType) , np.array(pred_X,
                dtype=self.dataType)
90
91      # 对训练样本序号随机分组
92      def getTrainRanges(self, miniBatchSize):
93          rangeAll = self.sample_range
94          random.shuffle(rangeAll)
95          rngs = [rangeAll[i:i + miniBatchSize] for i in
                range(0, len(rangeAll), miniBatchSize)]
96          return rngs
97
98      # 获取训练样本范围对应的训练数据和验证标准
99      def getTrainDataByRng(self, rng):
100         xs = np.array([self.x[sample] for sample in rng],
                self.dataType)
101         values = np.array([self.y[sample] for sample in
                rng])
102         return xs, values
103
104     # 获取验证样本，不打乱，用于显示连续曲线
105     def getValData(self):
106         return self.x,self.y
```

模型训练时，只要初始化数据预处理类，就可以用上述已实现的方法分批调用训练数据。

```
1 # 初始化处理类，生成数据对象
2 seqData = Hs300Data(hs300_file,Params.DTYPE_DEFAULT)
3 # 获取数据的mini_batch分组结果
4 dataRngs=seqData.getTrainRanges(Params.MINI_BATCH_SIZE)
```

接下来，就可以在训练迭代中，循环获取随机分组的训练数据及其对应标注送入模型。

```
1 x, y_ = seqData.getTrainDataByRng(dataRngs[batch])
```

8.7.2 模型构建

将训练数据中 64 个交易日的数据划为一组，分组后随机打乱组间次序，每组数据内部的时序信息仍然保留，用于特征捕获；每一组作为一个样本，用随机组合的 32 个样本组成一个 mini-batch，送入模型。

模型中，每个 LSTM 单元包含 64 个时间步，为每个时间步设置 128 个隐藏节点，构造三层 LSTM 结构，最后经过全连接层处理，得到 20 个一维结果数据作为模型输出，与未来 20 个交易的真实指数一起，计算平方误差损失，用于模型训练。

这个三层 LSTM 模型中各处理层的结构和参数定义如下：

- 输入层，输入数据 mini-batch 为 32，每段序列 64 个时间步，输入数据的维度 32×64。
- LSTM 第一层, 同样有 64 个时间步，每个时间步 128 个隐藏节点，分别看参数量：$\boldsymbol{W}_h$ 为 $128\times4\times128+4\times128 = 66048$；$\boldsymbol{W}_x$ 为 $1\times4\times128+4\times128 = 1024$; 第一层参数合计 67072 个。
- 第二层结构和第一层一致，每个时间步上的输入维度为 128×128, 参数量更多：$\boldsymbol{W}_h$ 为 $128\times4\times128+4\times128 = 66048$；$\boldsymbol{W}_x$ 为 $128\times4\times128+4\times128 = 66048$，第二层参数合计 132096 个。
- 第三层的结构、参数量和第二层一致，参数合计 132096 个；在目标场景下，只取最后一个节点的输出。
- 最后一个全连接层，输入数据维度 32×128，输出数据维度 32×20, 参数量 $128 \times 20 + 20 = 2580$ 个。

各层参数量如表8.1所示。

表 8.1　长短时记忆网络 LSTM 三层结构参数量

处理层	$\boldsymbol{W}_h$	$\boldsymbol{W}_x$	参数合计
输入	\	\	无
LSTM-1	66048	1024	67072
LSTM-2	66048	66048	132096
LSTM-3	66048	6604	132096
FC	$\boldsymbol{W}$:2560	b:20	2580

LSTM 的算法采用先计算加权变换，再拆分结果送给不同逻辑门的方式，虽然写法不同，参数的数量是一样的。

这个预测场景的学习策略优化算法，同样**不借助深度学习框架**，复用之前章节中实现的 **Adagrad 算法**。

8.7.3　分析结果

经过 3000 轮迭代训练，模型的验证 Loss 曲线趋于稳定，如图8.3所示。

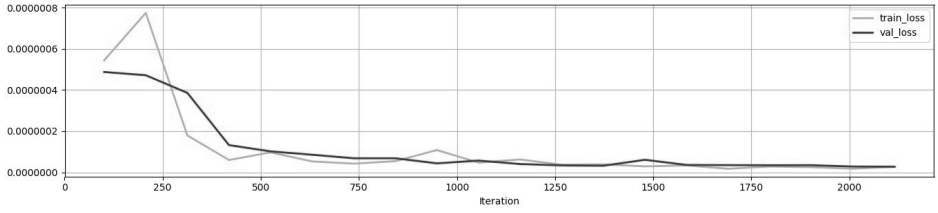

图 8.3　Loss 曲线

得到的趋势曲线，与真实指数变化的曲线做对比，参见图8.4。

也可以用过去最后 64 个交易日的数据，试算未来 20 个交易日的指数走势。

然而，即使不考虑过拟合问题，对一个受诸多因素影响，处于多方动态博弈之下的市场，即使有模型能够捕获市场上有价值的特征，这个模型也必然是动态的，且条件特征不会仅仅取决于历史有效交易日的成交数据，因此这个例子不建议作为投资决策的参考。

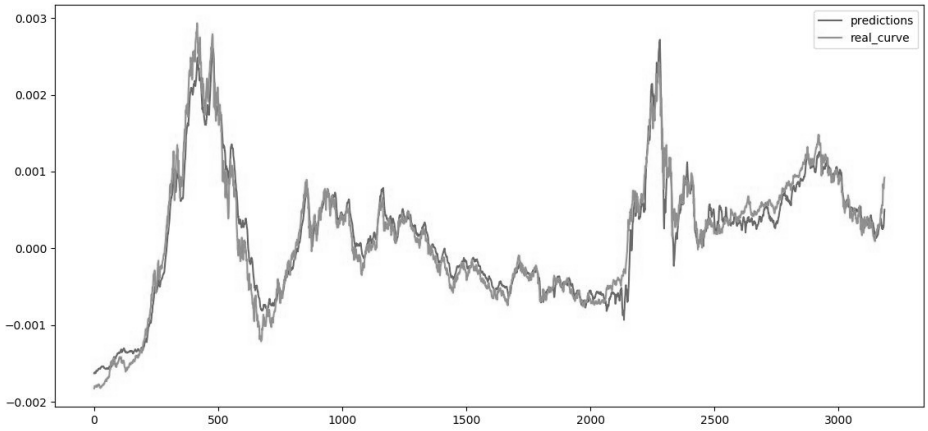

图 8.4　沪深 300 价格指数，趋势拟合后曲线对比

回到算法本身，观察图 8.4，训练得到的模型参数，对指数历史走势的规律，有了一定的拟合能力。

8.8　小结

这一章，首先介绍了 Vanilla RNN 模型处理长程序列数据时所遇到的挑战。以此为动机，引入基础 RNN 的衍生模型。作为 Vanilla RNN 的一种较为成功的变体，长短时记忆网络（LSTM）模型较好地缓解了在长时间步下的梯度传递问题。在本章的最后，以沪深 300 指数分析作为目标问题，使用所构建的三层 LSTM 模型，对指数的历史走势规律做了特征提取，验证了 LSTM 模型对长程序列数据的处理效果。

下一章继续介绍同样采用逻辑门机制的另一种 RNN 变体，即 GRU 模型算法。GRU 采用相对简洁的模型结构，能以较少的算力消耗，得到与 LSTM 相近的效果。

拓展阅读

Christopher Olah 分享对 LSTM 模型设计思想的理解，用分解图的方式描述了 LSTM 前向传播计算的步骤。

Understanding LSTM Networks

http://colah.github.io/posts/2015-08-Understanding-LSTMs

多伦多大学的神经网络与机器学习课程 (csc321) 第 15 讲，分析了 RNN 梯度传递问题的成因及其几种缓解办法。
Intro to Neural Networks and Machine Learning, Lecture 15: Exploding and Vanishing Gradients.
http://www.cs.toronto.edu/~rgrosse/courses/csc321_2018

参 考 文 献

[1] Bengio, Yoshua, Simard, Patrice, and Frasconi, Paolo. **Learning long-term dependencies with gradient descent is difficult.** IEEE Transactions on Neural Networks, 1994, 5(2):157–166.

[2] Sepp Hochreiter, Jürgen Schmidhuber. **Long Short Term Memory.** Neural Computation, 1997, 9(8):1735-1780.

[3] Klaus Greff, Rupesh K. Srivastava, Jan Koutn, Bas R Steunebrink, Jürgen Schmidhuber. **LSTM: A Search Space Odyssey**. arXiv:1503.04069v2, 2017.

第 9 章

双向门控循环单元（BiGRU）——情感分析

“合抱之木，生于毫末；九层之台，起于累土；千里之行，始于足下。”

——[春秋] 老子《道德经》

上一章讨论的长短时记忆网络（LSTM），引入逻辑门机制，缓解了长程时间步下的梯度传递问题。Vanilla RNN 模型还有另一个被广泛使用的成功变体：门控循环单元（Gated Recurrent Unit，GRU）。GRU 模型的结构比 LSTM 简洁，运算代价也较低，却能取得和前者接近的效果。

这一章，首先介绍 GRU 模型结构、前向和反向传播算法；然后**不借助深度学习框架**，实现这些算法，并应用于语言模型，再基于真实的电影评论数据集，实现情感分析来验证这个模型；验证过程中，通过 Dropout 算法，观察过拟合问题的缓解效果；最后使用双向算法把模型改进为 BiGRU 结构，进一步提高模型在目标问题上的训练效果。

9.1 目标问题：情感分析

社交平台上，每时每刻，都有人用内容含义丰富的文字，对服务、产品、公众人物或热点事件做出评价。

这一章，我们使用 IMDB（`http://ai.stanford.edu/~amaas/data/sentiment`）的公开数据集作为分析对象。这套数据集是互联网电影资料库公开的影评数据集，包括大量电影观众对两千余部电影做出的或褒或贬（positive/negative）的评价留言，共计五万条记录，其中每部影片的评价不超过 30 条，褒贬各半，如果随机划分的话，则只能对褒贬含义得到 50% 上下的识别正确率。

在数据集里，点评留言的情绪极性，不容易通过褒贬关键词来简单判断，以数据集里的两条影评为例。

带有褒扬含义的评价：

> This is the kind of film you want to see with a glass of wine, the fire on, and with your feet up. It doesn't require that much brain-power to follow, so is very good after a long day. I would say it is very unrealistic - if you expecting anything serious, then don't bother, but it is very funny. Just the thought that a businessman would go so far as to agree to live in a slum for a while, and then actually get to enjoy it… I would definitely recommend it.
>
> …

表现出负面情绪的评价：

> It was a Sunday night and I was waiting for the advertised movie on TV. They said it was a comedy! The movie started, 10 minutes passed, after that 30 minutes and I didn't laugh not even once. The fact is that the movie ended and I didn't get even one chance to laugh. PLEASE, someone tickle me, I lost 90 minutes for nothing.
>
> …

面对这种分析场景，需要深度学习模型具备对长程信息自适应取舍的能力，结合整段评价的上下文来判别点评留言的褒贬情绪。

9.2 第一个挑战：模型的运算效率

在序列分析场景上，LSTM 较 Vanilla RNN 前进了一大步，每个 LSTM 时间步包含记忆单元和四个门控节点，通过四个门控节点。LSTM 实现了模块内部信息流“记忆”和“遗忘”权重的自适应控制。

这四个门控单元相应增加了模型的计算复杂度。在多层、多时间步叠加的结构下，如果需要使用大量输入数据做模型训练，就需要更强的算力支持，来确保训练的运算效率。

受 LSTM 模型的启发，Kyunghyun Cho 和他的同事在 2014 年提出另一种 RNN 变体：门控循环单元（GRU）。

GRU 的模型结构更为简洁，能以较低的算力消耗，得到与 LSTM 相近的效果。

9.3 GRU 模型的结构

同 LSTM 模型相比，GRU 结构简洁一些，只用了重置门 r_t 和更新门 z_t 两个门控节点，来计算各时间步的隐藏节点状态，GRU 模型单个时间步的内部结构参见图9.1。

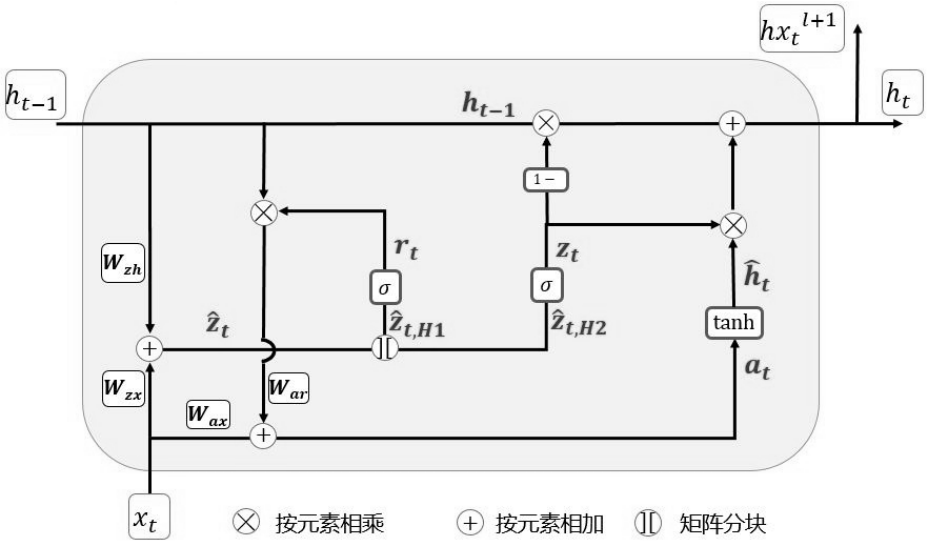

图 9.1 GRU 模型结构

重置门 r_t

决定是否忽略上一个时间步的隐状态 $\boldsymbol{h}_{t-1}$；如果激活权重降低，当前 GRU

时间步会更多地捕捉到短时间跨度的依赖信息，重置门激活趋于 0 时，当前时间步的输入 x_t 会更大程度影响隐状态 $\hat{h}_t$ 的输出，起到遗忘长时间跨度依赖的效果。

更新门 z_t

选择是否用新的候选隐状态 $\hat{h}_t$ 来更新输出隐状态 $\boldsymbol{h}_t$。GRU 中更新门的作用，类似 LSTM 模型里，输入门 i 和准单元状态 g 一起构成的记忆单元。如果 z_t 以较大权重激活向上的分支，则输入 $\boldsymbol{x}_t$ 对输出隐状态的影响下降，使模型更多地捕捉到长时间跨度的依赖信息，缓解了反向传播时在长程时间步下的梯度弥散问题；在极端情况下，向上的分支被完全激活，此时，从上一时间步的隐状态输出 $\boldsymbol{h}_{t-1}$ 到当前时间步隐状态输出 $\boldsymbol{h}_t$ 之间，形成信息的短路传递。

GRU 模型每个时间步各自具有重置门 r_t 和更新门 z_t，每个时间步的隐藏节点会学习捕捉不同时间步跨度下不同依赖信息的特征。

各个时间步的隐藏节点叠加在一起，共同根据数据流上的信息与结果的相关程度，自适应地学习到不同时间跨度下依赖信息的取舍权重，使输出更紧凑地表达信息。

9.4 GRU 前向传播算法

首先，用当前时间步的输入 $\boldsymbol{x}_t$ 和上一时间步的隐藏状态输出 $\boldsymbol{h}_{t-1}$，计算得到原始前馈输出 $\hat{\boldsymbol{z}}_t$：

$$\hat{\boldsymbol{z}}_t = \boldsymbol{W}_{zx}\boldsymbol{x}_t + \boldsymbol{W}_{zh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_z \tag{9.1}$$

然后把这个输出结果一分为二，即 $\hat{\boldsymbol{z}}_{t,H1}$ 和 $\hat{\boldsymbol{z}}_{t,H2}$，分别激活：

$$\boldsymbol{r}_t = \sigma(\hat{\boldsymbol{z}}_{t,H1}) \quad \text{（重置门）} \tag{9.2}$$

$$\boldsymbol{z}_t = \sigma(\hat{\boldsymbol{z}}_{t,H2}) \quad \text{（更新门）} \tag{9.3}$$

再使用当前时间步的重置门输出 $\boldsymbol{r}_t$ 和原始输入 $\boldsymbol{x}_t$ 一起计算另一个中间前馈输出：

$$\boldsymbol{a}_t = \boldsymbol{W}_{ax}\boldsymbol{x}_t + \boldsymbol{W}_{ar}(\boldsymbol{h}_{t-1} \otimes \boldsymbol{r}_t) + \boldsymbol{b}_a \tag{9.4}$$

对中间前馈输出 $\boldsymbol{a}_t$ 做双曲正切激活：

$$\hat{\boldsymbol{h}}_t = \tanh(\boldsymbol{a}_t) \tag{9.5}$$

以更新门的输出作为开关，得到当前时间步的隐状态输出：

$$\boldsymbol{h}_t = \boldsymbol{h}_{t-1} \otimes (1 - \boldsymbol{z})_t + \boldsymbol{z}_t \otimes \hat{\boldsymbol{h}}_t \qquad \blacksquare \tag{9.6}$$

以上，是 GRU 前向传播算法的完整表达式。

GRU 的表达式比 LSTM 简化一些，然而在不同的论文和各类深度学习框架里会出现其他不同的写法，这些不同的写法容易给初学者造成困惑，也可能给从业者带来沟通上的误解，接下来对比分析几种常见但不同的表达，以便在后续章节对 GRU 算法予以准确且前后一致的实现。

9.5 GRU 前向传播表达式的其他写法

GRU 模型的计算有一些形式不同但效果等价的写法。

对式 (9.1)，同 Vanilla RNN 类似，也可以把 $\boldsymbol{W}_{zx}$ 和 $\boldsymbol{W}_{zh}$ 拼接为一个 $\boldsymbol{W}_z$ 矩阵，同时拼接上一层输入和前一时刻隐藏节点输出 $[\boldsymbol{x}_t, \boldsymbol{h}_{t-1}]$ ，再和权参 $\boldsymbol{W}_z$ 做仿射变换：

$$\boldsymbol{z}_t = \boldsymbol{W}_z[\boldsymbol{x}_t, \boldsymbol{h}_{t-1}] + \boldsymbol{b}_z \tag{9.7}$$

对式 (9.1)，还有另一种较常见的写法，把 $\boldsymbol{W}_{zx}$ 和 $\boldsymbol{W}_{zh}$ 拆分成四个权值参数矩阵，分别和 $\boldsymbol{x}_t$，$\boldsymbol{h}_{t-1}$ 做仿射变换，再拼接得到 $\hat{\boldsymbol{z}}_{t,H1}$，$\hat{\boldsymbol{z}}_{t,H2}$。

对式 (9.4)，较早提出 GRU 方法的论文（Cho *et al.*，2014）写为

$$\boldsymbol{a}_t = \boldsymbol{W}_{ax}\boldsymbol{x} + (\boldsymbol{W}_{ar}\boldsymbol{h}_{t-1}) \otimes \boldsymbol{r}_t + \boldsymbol{b}_a \tag{9.8}$$

实验证明（Chung *et al.*，2014），式 (9.4) 和式 (9.8) 两种方法的效果相同。

对式 (9.6)，也有另一种写法：

$$\boldsymbol{h} = \boldsymbol{h}_{t-1} \otimes \boldsymbol{z}_t + (1 - \boldsymbol{z})_t \otimes \hat{\boldsymbol{h}}_t \tag{9.9}$$

显然，由于模型自适应地学习合适的激活权重，训练和预测时，只要表达

式保持一致，式 (9.6) 和式 (9.9) 的训练结果是等价的。

本书 9.4 节所选用的计算方式和表达式书写，同样是为了便于观察反向传播算法的推导步骤。

9.6 GRU 反向传播算法

对多层叠加、多时间步的 GRU 模型，整个 GRU 单元的输出误差，是由沿时间和层次两个方向反向传播误差的叠加而成的，参见图9.2。

$$\delta \boldsymbol{h} = \delta \boldsymbol{h}_{t+1} + \delta \boldsymbol{h}_t^{l+1} \tag{9.10}$$

由矩阵的复合求导法则计算中间前馈输出的误差损失。

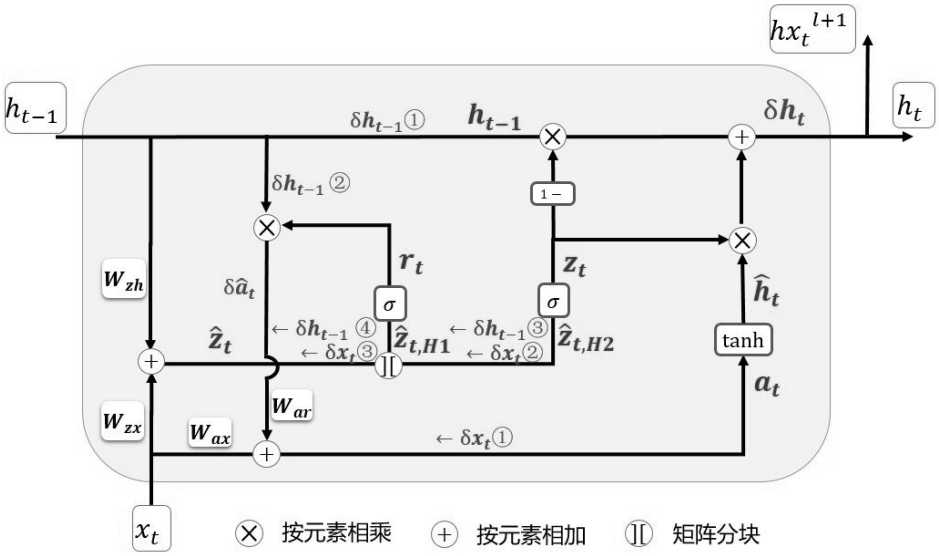

图 9.2 GRU 单个时间步的反向传播

$$\delta \boldsymbol{a} = \delta \boldsymbol{h} \otimes \boldsymbol{z} \otimes \tanh'(\boldsymbol{a}) \tag{9.11}$$

再继续推算这一步损失对应上一时间步隐状态输出的误差分量。

$$\delta \hat{\boldsymbol{a}} = \boldsymbol{W}_{ar}^{\mathrm{T}} \cdot \delta \boldsymbol{a} \tag{9.12}$$

$\delta \boldsymbol{z}$ 的误差来自上部和右侧，其中，来自上部的反向传播误差：

$$\delta \boldsymbol{z}_{\text{up}} = -\delta \boldsymbol{h} \otimes \boldsymbol{h}_{t-1} \tag{9.13}$$

来自右侧的反向传播误差：

$$\delta \boldsymbol{z}_{\text{right}} = \delta \boldsymbol{h} \otimes \hat{\boldsymbol{h}}_t \tag{9.14}$$

把来自两个方向的回传误差叠加起来，用于后续反向传播计算：

$$\delta \boldsymbol{z} = \delta \boldsymbol{h} \otimes (\hat{\boldsymbol{h}} - \boldsymbol{h}_{t-1}) \tag{9.15}$$

重置门的反向传播误差：

$$\delta \boldsymbol{r} = (\boldsymbol{W}_{ar}^{\mathrm{T}} \cdot \delta \boldsymbol{a}) \otimes \boldsymbol{h}_{t-1} \tag{9.16}$$

两部分误差，拼接得到完整误差矩阵：

$$\delta \hat{\boldsymbol{z}} = [\delta \boldsymbol{r} \otimes \text{sigmoid}'(\boldsymbol{z}_{H1}), \delta \boldsymbol{z} \otimes \text{sigmoid}'(\boldsymbol{z}_{H2})] \tag{9.17}$$

隐状态 $\delta \boldsymbol{h}_{t-1}$ 误差，来自四个部分的叠加：

$$\delta \boldsymbol{h}_{t-1} = \delta \boldsymbol{h} \otimes (1 - \boldsymbol{z}) + (\boldsymbol{W}_{ar}^{\mathrm{T}} \cdot \delta \boldsymbol{a}) \otimes \boldsymbol{r} + \boldsymbol{W}_{zh,H1}^{\mathrm{T}} \cdot \delta \hat{\boldsymbol{z}}_{t,H1} + \boldsymbol{W}_{zh,H2}^{\mathrm{T}} \cdot \delta \hat{\boldsymbol{z}}_{t,H2} \tag{9.18}$$

输入 $\delta \boldsymbol{x}$ 误差，同样来自三个部分的叠加：

$$\delta \boldsymbol{x} = \boldsymbol{W}_{ar}^{\mathrm{T}} \cdot \delta \boldsymbol{a} + \boldsymbol{W}_{zx,H1}^{\mathrm{T}} \cdot \delta \hat{\boldsymbol{z}}_{t,H1} + \boldsymbol{W}_{zx,H2}^{\mathrm{T}} \cdot \delta \hat{\boldsymbol{z}}_{t,H2} \tag{9.19}$$

即可继续推出 GRU 单元各参数梯度。

重置门权参的梯度：

$$\nabla_{\boldsymbol{W}_{ax}} E = \delta \boldsymbol{a}\ \boldsymbol{x}^{\mathrm{T}} \tag{9.20}$$

$$\nabla_{\boldsymbol{W}_{ar}} E = \delta \boldsymbol{a} (\boldsymbol{r} \otimes \boldsymbol{h}_{t-1})^{\mathrm{T}} \tag{9.21}$$

重置门偏置参数的梯度：

$$\nabla_{b_a} E = \delta \boldsymbol{a} \tag{9.22}$$

中间前馈输出的参数梯度：

$$\nabla_{\boldsymbol{W}_{zx}} E = \delta \hat{\boldsymbol{z}}\ \boldsymbol{x}^{\mathrm{T}} \tag{9.23}$$

$$\nabla_{\boldsymbol{W}_{zh}} E = \delta \hat{\boldsymbol{z}} \boldsymbol{h}_{t-1}^{\mathrm{T}} \tag{9.24}$$

重置门偏置参数的梯度：

$$\nabla_{\boldsymbol{b}_z} E = \delta \hat{\boldsymbol{z}} \qquad \blacksquare \tag{9.25}$$

以上得到 GRU 单元一个时间步上反向传播的全部误差传递和参数更新梯度。GRU 单元一个层级上各个时间步在同一个传播方向上共享权值和偏置参数，把各时间步上的梯度加总，即可得到该层 GRU 最终的参数梯度。

9.7 GRU 算法实现

基于上面的推导，可以**不借助深度学习框架**，实现 GRU 的前向和反向传播算法，并封装为多层、多时间步的 GRU 模型，用于序列分析。

9.7.1 单时间步的前向计算

在单个时间步结构里，用当前时间步的输入 $\boldsymbol{x}_t$ 和上一时间步的隐藏状态输出 $\boldsymbol{h}_{t-1}$ 计算得当前时间步的隐状态输出 $\hat{\boldsymbol{z}}_t$：

```
# GRU单个时间步前向传播
def gru_step_forward(x, prev_h, Wzx, Wzh, bz, Wax, War, ba
    ):
    H = prev_h.shape[1]
    # 原始前馈输出
    z_hat = Tools.matmul(x, Wzx) + Tools.matmul(prev_h,
        Wzh) + bz
    # of shape(N,H)
    # 重置门
    r = Tools.sigmoid(z_hat[:, :H])
```

```
    # 更新门
    z = Tools.sigmoid(z_hat[:, H:2 * H])
    # 另一路前馈输出
    a = Tools.matmul(x, Wax) + Tools.matmul(r * prev_h,
        War) + ba
    # 隐状态输出
    next_h = prev_h * (1. - z) + z * np.tanh(a)
    # 缓存反向传播用的中间变量
    cache = (x, prev_h, Wzx, Wzh, Wax, War, z_hat, r, z, a
        )
    # 返回当前时间步的隐状态输出，中间计算结果用于反向传播
        计算
    return next_h, cache
```

9.7.2 实现单时间步的反向传播

单时间步的反向传播需要用到前向传播时保留的中间变量，和上一时间步回传误差共同计算。

```
# GRU单个时间步的反向传播
def gru_step_backward(dnext_h, cache):
    # 加载前向传播时保留的中间结果
    x, prev_h, Wzx, Wzh, Wax, War, z_hat, r, z, a = cache
    # 获取当前时间步输入的维度xiang N, D = x.shape
    # 隐藏节点维度
    H = dnext_h.shape[1]
    # 拆分更新门输出，用于BP计算
    z_hat_H1 = z_hat[:, :H]
    z_hat_H2 = z_hat[:, H:2 * H]
    tanha = np.tanh(a)
    dh = dnext_h
    da = dh * z * (1. -{\rm tanh}a ** 2)
    dh_prev_1 = dh * (1. - z)
    # 计算更新门的两路BP误差
    dz_hat_2 = dh * (tanha - prev_h) * (z * (1. - z))
    dhat_a = Tools.matmul(da, War.T)
    # 重置门BP误差
```

```
    dr = dhat_a * prev_h
    dx_1 = Tools.matmul(da, Wax.T)
    dh_prev_2 = dhat_a * r
    dz_hat_1 = dr * (r * (1. - r))
    # 拼接更新门的误差张量
    dz_hat = np.hstack((dz_hat_1, dz_hat_2))
    dx_2 = Tools.matmul(dz_hat_1, Wzx[:, :H].T)
    dh_prev_3 = Tools.matmul(dz_hat_1, Wzh[:, :H].T)
    dx_3 = Tools.matmul(dz_hat_2, Wzx[:, H:2 * H].T)
    dh_prev_4=Tools.matmul(dz_hat_2, Wzh[:, H:2 * H].T)
    # 合并隐藏状态的四路BP误差
    dprev_h = dh_prev_1 + dh_prev_2 + dh_prev_3 +
        dh_prev_4
    # 合并当前时间步输入环节的三路BP误差
    dx = dx_1 + dx_2 + dx_3
    # 权参张量的BP梯度
    dWax = Tools.matmul(x.T, da)
    dWar = Tools.matmul((r * prev_h).T, da)
    # 偏置项BP梯度
    dba = np.sum(da, axis=0)
    dWzx = Tools.matmul(x.T, dz_hat)
    dWzh = Tools.matmul(prev_h.T, dz_hat)
    dbz = np.sum(dz_hat, axis=0)
    # 返回当前时间步的反向传播误差和参数更新梯度
    return dx, dprev_h, dWzx, dWzh, dbz, dWax, dWar, dba
```

实现了单层单时间的 GRU 算法，就可以参考 LSTM 模型多层多时间步的逻辑，只需要移除对隐藏单元状态 c_t 的前向传递，以及单元状态反向传播的误差分量叠加，即可实现 GRU 模型多层、多时间步的算法。此处不再占用篇幅，留给读者思考和尝试实现。

9.8 用 GRU 模型进行情感分析

接下来，用 GRU 算法搭建情感分析模型，对目标问题：真实的电影评价数据，做语言的情感极性分析。

9.8.1 数据预处理

评论数据在送入模型之前，需要先按照以下步骤做预处理。

1. 去除停用词。停用词（stop word）是句子中一些高频出现的，但是在当前场景下对整体情绪极性表达影响较小的词汇，比如冠词 “a, an, the”，人称代词 “you，I，he，she”，指示代词 “this, that,these,those” ，以及较常见的介词 “in，at，on，of”、连词 “and，or，either，neither” 等。去除这些停用词，有助于提高处理效率。已经有大量的工作对各个语种的停用词做了汇总整理，并且开放出来以供使用。当前目标场景下，使用 nltk 开源库`http://www.nltk.org`中收集的英文停用词表，对电影评论的语料做停用词过滤。

2. 语句填充（padding）。我们实现的 GRU 模型，可以定义不同长度的时间，对来自同一集合的目标数据，预先定义了固定时间步长度的模型来做分析；然而数据集中每条点评留言字数长短不一，可以根据留言字数的直方图分析，观察语料中各条评论记录字数的分布，权衡模型训练效率，希望从较短的句子长度中捕捉到的尽可能达意的情绪极性特征，以此来选择合适的模型时间步长度。在这个场景下，定义模型每一层时间步长度为 400 个词；然后对每一条评论语句，截去超长的部分；对于不到固定长度的句子，用填充字符补足到定长。用这种方式实现对不定长语料的分析。

3. 获取目标语句的词向量（word vector）。词向量，是词汇的向量形式表达。这一步，通过词嵌入矩阵把评价文字中的词汇，通过特定的映射关系，转换为编码表示的词向量，把评价留言记录转换为词向量，使得预处理后的输入数据既尽可能地保留语义信息，又能被模型方便地处理。由词汇训练获得词向量的原理和算法，是另一个值得探索的领域，基于本书目前已经讨论验证过的基础算法，可以收集语料，在深度学习框架上训练自己的词嵌入矩阵。由于本章重点是 GRU 模型底层算法的原理与实践，对于词向量映射的原理和算法，此处不再展开。这个案例中，采用斯坦福大学 GloVe 开源项目，预训练并公开的 50 维 40 万英文词嵌入矩阵`https://nlp.stanford.edu/projects/glove`，映射得到点评留言的词向量。

4. 数据集划分。把转换为词向量的语句集合，随机打乱次序，再按 9 : 1 的比例划分为训练集和测试集两个独立集合。

9.8.2 构建情感分析模型

用前文实现的 GRU 模型，搭建一个三层叠加的门控循环神经网络，每层 400 个时间步，每个时间步 32 个隐藏节点，在训练集中随机抽取 mini-batch 送入 GRU 模型。GRU 模型的输出，经过一个全连接层，进一步映射到褒贬（positive/negative）两个输入类别上。通过 softmax 把输出映射为概率分布，再和正确分类标注一起，计算交叉熵损失。反向传播训练时，采用之前实现的 Adam 优化学习策略训练模型参数。

这个场景下，只需要对语言情感极性做两个类别的划分，所以也可选用 logistic 二分类回归代替 Softmax 处理；此处选用 Softmax 方法，是为了使模型更加灵活，可不加修改，直接兼容支持更多类别的情感极性划分场景。

三层 GRU 模型的结构定义及各处理层的参数如下。

- 输入数据，每个样本序列 400 个时间步，以随机选取的 16 个样本组成一个 mini-batch 送入模型，完成一次训练，输入数据的维度 16×400。
- 第一层 GRU，同样包含 400 个时间步，每个时间步 32 个隐藏节点。参数量 $\boldsymbol{W}_{zh}$ 为 $32 \times 2 \times 32 + 2 \times 32 = 2112$；$\boldsymbol{W}_{ar}$ 为 $32 \times 32 + 32 = 1056$; $\boldsymbol{W}_{zx}$ 为 $1 \times 2 \times 32 + 2 \times 32 = 128$; $\boldsymbol{W}_{ax}$ 为 $1 \times 32 + 32 = 64$ ；第一个 GRU 层参数合计 3360 个。
- 第二层 GRU 结构与第一层一样，经过前层处理，本层输入维度成为 32×400，参数较第一层有所增加：$\boldsymbol{W}_{zh}$ 为 $32 \times 2 \times 32 + 2 \times 32 = 2112$；$\boldsymbol{W}_{ar}$ 为 $32 \times 32 + 32 = 1056$；$\boldsymbol{W}_{zx}$ 为 $32 \times 2 \times 32 + 2 \times 32 = 2048$；$\boldsymbol{W}_{ax}$ 为 $32 \times 32 + 32 = 1056$ ；第二个 GRU 层参数合计 6272 个。
- 第三层 GRU 结构和参数量都与第二层一致，当前目标场景下，只取最后一个时间步的输出。
- 最后一个全连接层，输入数据维度为 16×400，输出数据维度为 16×2, 参数量为 $400 \times 2 + 2 = 802$ 个。

各层参数量参见表 9.1。

表 9.1　三层结构门控循环单元网络结构处理层和参数量

处理层	$\boldsymbol{W}_{zh}$	$\boldsymbol{W}_{ar}$	$\boldsymbol{W}_{zx}$	$\boldsymbol{W}_{ax}$	参数合计
输入	\	\	\	\	无
GRU-1	2112	1056	128	64	3360
GRU-2	2112	1056	2048	1056	6272
GRU-3	2112	1056	2048	1056	6272
FC	\	\	$\boldsymbol{W}$:800	$\boldsymbol{b}$:2	802

9.9　首次验证

把预处理过的影评数据送入模型后，经过 5000 次迭代训练，发现模型在训练数据集和独立测试集上表现，出现了比较大的差异，参见图9.3。

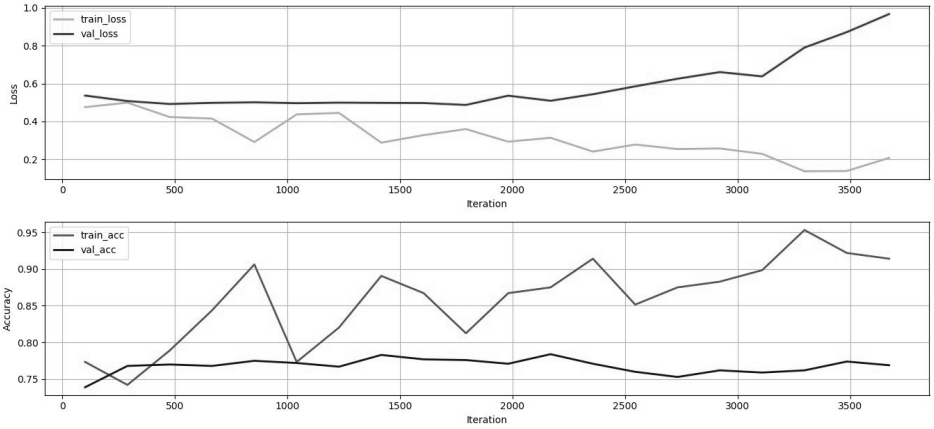

图 9.3　首次验证：三层 GRU 模型损失曲线, 出现了明显的过拟合

在训练数据集上，误差损失不断降低，表明模型在逐步收敛，然而在独立测试集上的验证损失，经过一定迭代训练后，不仅没有和训练损失一起逐步下降，反而呈现出上升的趋势，两条损失曲线在 1500 次迭代训练后，终于“分道扬镳”。验证数据集上的分类正确率达到 75% 以后，也不进反退。

这个现象说明，模型在训练数据集上能够收敛，然而泛化误差较大，出现了严重的**过拟合**（over-fitting）。

9.10　第二个挑战：序列模型的过拟合

发生了过拟合，说明模型在训练集上捕捉到了不具一般性的特征，因而在独立验证集上表现出较大的泛化误差。面对过拟合问题，一种方法是通过之前章节介绍过的 L_2 正则化方法，增强模型的泛化能力来缓解。

在这个情感分析的场景下，我们尝试用另一种有效的正则化方法：Dropout 正则化。

9.11　Dropout 正则化

使用总量不够大的数据集来训练神经网络的话，训练所得参数，在独立的验证数据集上往往效果不佳。Dropout 方法处理这种“过拟合”问题的方法是：每轮训练时，随机丢弃一部分隐藏节点的输出，以此来改善模型的泛化能力，参见图9.4。

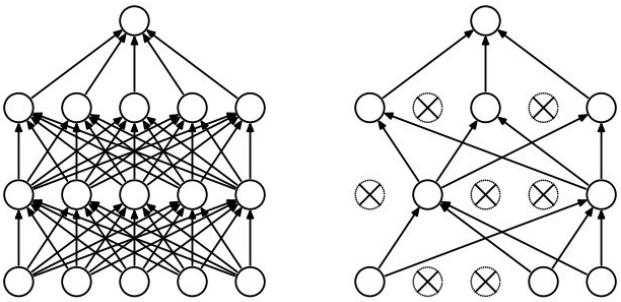

图 9.4　Dropout 方法，左图是标准的神经网络，右图是应用 Dropout 方法后的结构。图片来自参考文献

通过“随机部分丢弃”这个简单的措施，Dropout 方法能够减少各隐藏节点的复杂共适应（prevents complex co-adaptations ）效应，使某个节点所捕捉的数据特征不依赖其他隐藏节点的特定状态，以此阻止个别隐藏节点出现不合理的高权重。从这个角度看，Dropout 方法和 L2 正则化使用了相近的原理来缓解过拟合问题。

9.11.1　Dropout 前向传播算法

假设某个隐藏层 a^l 的节点输出，以概率 p 被保留，即以概率 $1-p$ 随机丢弃该层部分节点的原始输出。这个处理本质上是对该层的每个隐藏节点输出，按照既定概率做随机筛选。

使用 Dropout 处理节点输出之前，首先生成一个 mask 筛选向量，其每个分量都是一个独立的伯努利随机变量（independent Bernoulli random variable）：

$$\text{mask}^l \sim \text{Bernoulli}(n, p) \tag{9.26}$$

可以用 NumPy 的随机二项分布方法 binomial() 生成这个 mask 筛选向量：

$$\text{mask}^l = \text{np.binomial}(n = 1, p) \tag{9.27}$$

当 $n = 1$ 时，伯努利实验得到的随机二项分布就成了 (0-1) 分布。每个随机变量有概率为 p 的机会等于 1；通过 mask 向量，就可以随机筛选隐藏节点的原始输出：

$$a^l_{\text{dropout}} = \text{mask}^l \otimes a^l \tag{9.28}$$

Dropout 方法只在模型训练的前向计算中进行。推理预测的时候，不再做 Dropout 正则化处理。为了让训练和预测的数值输出，保持相同的尺度（L_1 范数）。训练过程中，对节点输出做好随机筛选处理之后，需要再用保留概率 p, 对处理后的输出做数值放大。

$$a^l_{\text{final}} = \frac{1}{p} a^l_{\text{dropout}} \qquad \blacksquare \tag{9.29}$$

处理后的最终结果 a^l_{final}，作为 Dropout 输出，传递给下一层；本轮的 mask^l 筛选向量保留下来，反向传播时直接复用。

9.11.2 Dropout 反向传播算法

模型中被随机丢弃节点的输出值不影响本轮训练的输出。误差回传的时候，误差矩阵对应丢弃输出的位置将元素值设置为 0，只要把上一层回传的误差矩阵和本轮预先保留的 mask^l 筛选向量做 Hadamard 乘积，即可实现这一步骤：

$$\delta_{\text{dropout}} = \text{mask}^l \otimes \delta \tag{9.30}$$

基于上述原理，可以**不借助深度学习框架**，实现 Dropout 算法。

```
# GRU单个时间步前向传播
def gru_step_forward(x, prev_h, Wzx, Wzh, bz, Wax, War, ba
    ):
    H = prev_h.shape[1]
    # 原始前馈输出
    z_hat = Tools.matmul(x, Wzx) + Tools.matmul(prev_h,
        Wzh) + bz
    # of shape(N,H)
    # 重置门
    r = Tools.sigmoid(z_hat[:, :H])
    # 更新门
    z = Tools.sigmoid(z_hat[:, H:2 * H])
    # 另一路前馈输出
    a = Tools.matmul(x, Wax) + Tools.matmul(r * prev_h,
        War) + ba
    # 隐状态输出
    next_h = prev_h * (1. - z) + z * np.tanh(a)
    # 缓存反向传播用的中间变量
    cache = (x, prev_h, Wzx, Wzh, Wax, War, z_hat, r, z, a
        )

    return next_h, cache
```

前向计算时，保留 mask，用于反向传播时把误差矩阵对应丢弃输出位置的元素值设置为 0。

```
delta = delta_ori * dropoutMask
```

9.11.3 Dropout Rate 的选择

Dropout 正则化方法为模型引入了一个新的超参数 p。显然若 $p = 1$，等于不做 Dropout 处理；而较小的 p 参数，意味着将有更多节点输出被随机丢弃掉（Dropout）。

这个参数的取值视数据场景、分析任务和模型选择而定。在输入层，这个参数值取决于输入数据的类型，对视频图像或语音片段这类实值输入（real-

valued inputs），可以设置为 0.8。在隐藏层，可以结合隐藏节点数 $\boldsymbol{n}$（number of hidden units）来设定；若 $\boldsymbol{n}$ 较大，过大的 Dropout 丢弃率 $(1-p)$，往往会拖慢训练速度，而且导致欠拟合（under-fitting）发生；反过来，过小的丢弃率可能无法有效缓解过拟合（over-fitting）。

最早提出 Dropout 方法的 G.E.Hinton 团队建议，这个参数在 0.5~0.8 之间取值（Srivastava *et al*., 2014）。

9.12 再次验证：GRU+Dropout

对上面搭建的 GRU 情感分析模型，添加 Dropout（p=0.5），再次基于 IMDB 数据集做模型验证。

模型在独立测试集上的损失曲线，逐步下探；在这个场景下，**Dropout 缓解过拟合的作用，有效且直观**。参见图9.5。

只使用一半（24000 条）评价数据，经过 10 轮迭代训练，独立验证集上的推理预测正确率稳定在 80% 以上。

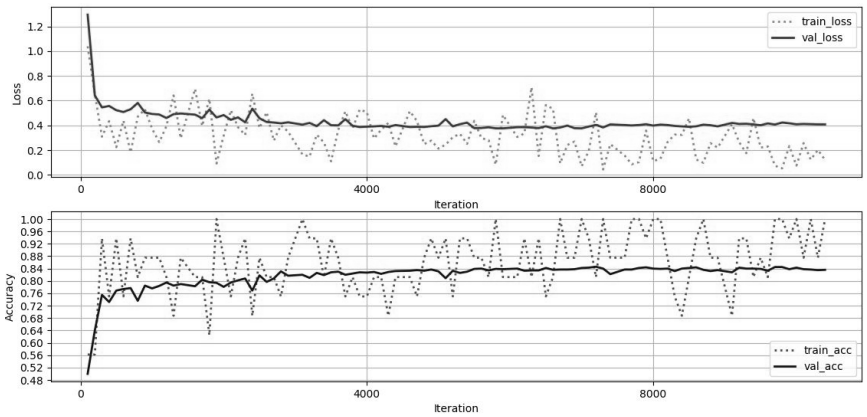

图 9.5 第二次验证：GRU+Dropout 损失曲线

在这个场景下，还可以采取下列数据增强手段得到更好的训练结果：

- 之前两轮的训练，只用到了一半影评数据；可以把尚未使用的剩余一半 IMDB 评价记录，一起送入模型参与训练，将训练样本扩大一倍。
- 将 GRU 模型的时间步参数定义从 400 调整到 500，同时增加输入语句填充（padding）的长度，以此送入更多的语义给模型来训练参数。
- 在数据预处理环节，换用更高维度的词嵌入矩阵，重新映射得到语义信息

更丰富的词向量送入模型。或者不使用第三方预训练的词嵌入矩阵，通过之前章节已经实现的基础算法，在深度学习框架上构建自己的词嵌入矩阵训练模型，针对 IMDB 影评数据这个特殊场景，定制训练出更适用的词嵌入矩阵，用来获取评价记录的词向量。

经过以上任意一种或多种手段对输入数据做增强，再送入这个情感分析模型进行训练，可以进一步得到更高的正确率。

回到算法的本身,我们先不通过上述数据增强手段,仅从本章的主题:“GRU 算法”出发，尝试做算法改进来提高模型效果。

9.13　第三个挑战：捕捉逆序信息

在电影评论这个场景下，作为分析目标的语言数据，表达的信息应该是前后呼应的，语义上既有自前向后的推进，也存在照应后文的伏笔。如果能够在正向和逆向两个方向上（Bidirectional）捕捉语义的序列特征，则应该可以识别出更深层的情感极性特征。

现在从算法入手，仍然**不借助深度学习框架**，把模型改造为**双向门控循环单元（BiGRU）**，同时捕捉逆序相关的语义信息。

9.14　双向门控循环单元（BiGRU）

在语言场景下，语义信息既有上下文正向关联，也有后文呼应前文的情况。对输入的语句序列数据，要捕捉更丰富的特征，除了增加堆叠层次外，还可以建立**双向门控循环单元 BiGRU** 模型，加入**逆向推理**机制，通过捕捉逆序特征，增强模型的表达能力。双向门控循环单元组合正、反向输入的方式与双向 RNN 模型类似，如图9.6所示。

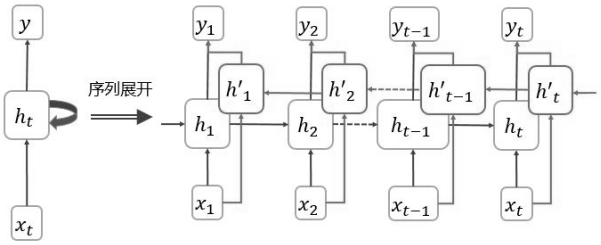

图 9.6　双向 RNN 模型

基于前面实现的单向模型，可以很容易地把多时间步的单向 GRU 的输出

改为两个方向上，分别得到多时间步的隐状态输出 $\boldsymbol{h}, \boldsymbol{h}'$ 的双向门控循环单元 BiGRU。

把两个隐状态输出，在隐藏节点维度拼接在一起：

$$\boldsymbol{y} = [\boldsymbol{h}, \boldsymbol{h}'] \tag{9.31}$$

仍然经过全连接层，进一步映射到褒贬（positive/negative）两个输入类别上，然后通过 Softmax 函数把输出映射为概率分布，再和正确分类标注一起，计算交叉熵损失。

损失回传到 BiGRU 层，同样在隐藏节点维度一分为二，得到正反双向误差损失：

$$\boldsymbol{\delta} \Longrightarrow \delta\boldsymbol{h}, \delta\boldsymbol{h}' \tag{9.32}$$

复用已实现的 GRU 反向传播算法，学习得到 BiGRU 在左右两个方向上的参数梯度。

如果 BiGRU 之前尚有前置处理层，则把左右两个方向上的误差损失叠加在一起，得到误差矩阵，继续反向传递：

$$\delta\boldsymbol{x} = \delta\boldsymbol{x}_f + \delta\boldsymbol{x}_b \qquad \blacksquare \tag{9.33}$$

在这个情感分析场景下，模型里的三层 BiGRU 是数据输入之后的首个处理层，不需要继续逆向传递误差损失，可不用获取最后一步计算的输出。

9.15 第三次验证：BiGRU+Dropout

使用上述算法，实现三层双向循环结构（BiGRU），仍然用 Dropout（p=0.5），基于 IMDB 数据集做第三次模型验证。

在相同同的超参数设置之下，较单向结构，双向结构模型收敛得更快，正确率也明显提高。仅使用一半的数据做模型训练，6 轮迭代训练后，尽管小批量训练集上的训练损失和正确率仍有起伏，独立验证集上损失却趋于稳定，验证集上的正确率稳定在 82% 以上, 参见图9.7。

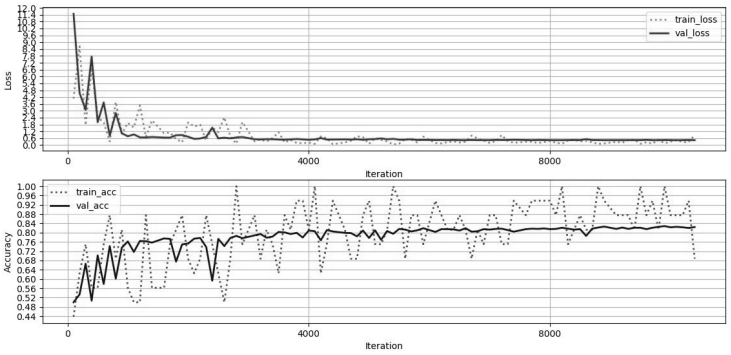

图 9.7 第三次验证：BiGRU+Dropout 损失曲线和正确率曲线

9.16 小结

循环神经网络 vanilla RNN 和主要变体**长短时记忆网络 LSTM**，以及**门控循环单元 GRU**，是神经网络模型中用于序列分析的重要一族。

至此，我们完成了循环神经网络 RNN、LSTM、GRU 模型，以及多层双向结构，前向和反向传播全部核心算法的推导与实现，并使用三种模型对各自的目标数据完成了**时序分析**、**指数分析**和**语言情感分析**，验证了序列模型对此类场景的强大处理能力。

参 考 文 献

[1] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Fethi Bougares, Holger Schwenk, Yoshua Bengio. **Learning phrase representations using rnn encoder-decoder for statistical machine translation.** arXiv:1406.1078v3, 2014.

[2] Junyoung Chung, Çaglar Gülçehre,KyungHyun Cho, Yoshua Bengio:**Empirical Evaluation of Gated RecurrentNeural Networks on Sequence Modeling**. arXiv:1412.3555v1, 2014.

[3] Andrew L Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, Christopher Potts.**Learning Word Vectors for Sentiment Analysis**.The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, 2011, 1: 142-150.

[4] Jeffrey Pennington, Richard Socher, Christopher D. Manning. **GloVe: Global Vectors for Word Representation**, 2014.

[5] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, Ruslan R. Salakhutdinov, **Improving neural networks by preventing co-adaptation of feature detectors**. arXiv:1207.0580, 2012.

[6] Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex,Sutskever, Ilya, and Salakhutdinov, Ruslan. **Dropout: A simple way to prevent neural networks from overfitting**. J. Mach. Learn. Res. 2014, 15(1):1929–1958.

[7] Alex Graves and Jurgen Schmidhuber. **Framewise phoneme classification with bidirectional lstm and other neural network architectures**. Neural Networks, 2005, 18(5):602–610.

[8] Schuster, M. and Paliwal, K. K. **Bidirectional recurrent neural networks**. IEEE Transactions on Signal Processing, 1997, 45:2673–2681。

附录 A

向量和矩阵运算

向量（Vector） 有序数表，列表长度是向量的维度。

n **维列向量：** $\boldsymbol{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$

m **维行向量：** $\boldsymbol{a} = \begin{pmatrix} a_1 & a_2 & \cdots & a_m \end{pmatrix}$

向量的几何意义 向量可以看成空间中从原点出发的矢量，具有长度和方向两个基本特征。

矩阵（Matrix） 由 $m \times n$ 个数 $a_{ij}(i = 1, 2, ..., m;\ j = 1, 2, ..., n)$ 排成的 m 行 n 列数表称为 $m \times n$ 矩阵：

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

矩阵的几何意义 矩阵可理解为描述线性空间中的特定变换。

向量点积（dot product）运算 即在欧氏空间上的**内积（inner product）**运算。

设有 n 维行向量 $\boldsymbol{a}=\begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix}$

n 维列向量 $\boldsymbol{b}=\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$

其内积运算为

$$\begin{aligned} \boldsymbol{a}\cdot\boldsymbol{b} &= a_1b_1+a_2b_2+\cdots+a_nb_n \\ &= \sum_{i=1}^{n} a_ib_i \end{aligned} \tag{A.1}$$

向量内积的几何意义 内积可以理解为一个向量在另一个向量方向上投影的长度，与另一向量长度的乘积，结果是一个标量。

矩阵乘积（matrix multiplication）运算 $m\times s$ 矩阵 $\boldsymbol{A}$ 和 $s\times n$ 矩阵 $\boldsymbol{B}$ 的乘积 $\boldsymbol{C}=\boldsymbol{AB}$, $\boldsymbol{C}$ 是一个 $m\times n$ 矩阵, 其中：

$$c_{ij}=\sum_{k=1}^{s} a_{ik}b_{kj} \quad (i=1,2,\cdots,m;\ j=1,2,\cdots,n) \tag{A.2}$$

进行乘积运算的矩阵，$\boldsymbol{A}$ 的列数必须等于 $\boldsymbol{B}$ 的行数。

线性变换（Linear transformation） 列向量 $\boldsymbol{x}$ 左乘矩阵 $\boldsymbol{W}$, 得到的乘积 $\boldsymbol{Wx}$ ，可以理解为空间中的向量 $\boldsymbol{x}$，经过特定变换 $\boldsymbol{W}$，变换为 $\boldsymbol{y}$。

$$\boldsymbol{y}=\boldsymbol{Wx} \tag{A.3}$$

仿射变换（Affine transformation） 列向量 $\boldsymbol{x}$ 左乘矩阵 $\boldsymbol{W}$, 得到的乘积 $\boldsymbol{Wx}$ ，再与偏置向量 $\boldsymbol{b}$ 相加。

$$\boldsymbol{y}=\boldsymbol{Wx}+\boldsymbol{b} \tag{A.4}$$

仿射变换和数学意义上的线性变换不同，可以理解为空间中的向量 $\boldsymbol{x}$，先做一次原点不变的线性变换，再平移。

矩阵按位（element wise）运算 两个同型矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的逐元素计算，结果是一个新的同型矩阵。式中 $\bigcirc$ 代表逐位加、减、乘、除 $(\oplus, \ominus, \otimes, \oslash)$ 中的任意一种运算符，计算表达式可以展开为

$$\boldsymbol{A} \bigcirc \boldsymbol{B} = \begin{pmatrix} a_{11} \bigcirc b_{11} & a_{12} \bigcirc b_{12} & \cdots & a_{1n} \bigcirc b_{1n} \\ a_{21} \bigcirc b_{21} & a_{22} \bigcirc b_{22} & \cdots & a_{2n} \bigcirc b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} \bigcirc b_{m1} & a_{m2} \bigcirc b_{m2} & \cdots & a_{mn} \bigcirc b_{mn} \end{pmatrix} \tag{A.5}$$

其中按位乘积运算 $\boldsymbol{A} \otimes \boldsymbol{B}$ 也称为 Hadamard 乘积运算。

向量的范数（Vector Norm） n 维向量 $\boldsymbol{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$ 的L_1 **范数**为向量元素的绝对值之和：

$$||\boldsymbol{a}||_1 = |a_1| + |a_2| + \cdots + |a_n| = \sum_{i=1}^{n} |a_i| \tag{A.6}$$

向量 L_1 范数的几何意义是 n 维空间中向量的终点到原点之间的**曼哈顿距离（Manhattan Distance）**。

L_2 **范数**为向量元素平方和再开平方：

$$||\boldsymbol{a}||_2 = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2} = \sqrt{\sum_{i=1}^{n} a_i^2} \tag{A.7}$$

向量 L_2 范数的几何意义是 n 维空间中向量的终点到原点之间的**欧氏距离（Euclidean Distance）**。

附录 B

导数和微分

微分（differential） 直观意义是描述函数 $y = f(x)$ 的图像上，从一点 x_0 移动无穷小量 Δx，到另一点 x_1，自变量与函数值变化的幅度。自变量 x 与函数值 y 的微分分别记为 $\mathrm{d}x, \mathrm{d}y$ 。

导数（derivative） 直观意义是描述函数 $y = f(x)$ 所表示的曲线在某一点处切线的斜率，体现函数值变化快慢的速率，记为

$$f'(x) = \frac{\mathrm{d}y}{\mathrm{d}x} \tag{B.1}$$

一元复合函数求导的链式法则（chain rule） 设 $y = f(u)$,$u = \phi(x)$, 且 $f(u), \phi(x)$ 都可导，则一元复合函数 $y = f(\phi(x))$ 的导数为

$$y' = \frac{\mathrm{d}y}{\mathrm{d}u}\frac{\mathrm{d}u}{\mathrm{d}x} = f'(u)\phi'(x) \tag{B.2}$$

一元复合函数全导数（total-derivative）链式法则

设 $y = f(u_1(x), u_2(x), \cdots, u_n(x))$, 在对应点 $(u_1, u_2, \cdots, u_n)$ 具有连续偏导数，且函数 $u_1(x), u_2(x), \cdots, u_n(x)$ 均在点 x 可导，则复合函数在点 x 处可导，导数为

$$\frac{\mathrm{d}f[(u_1(x), u_2(x), \cdots, u_n(x)]}{\mathrm{d}x} = \sum_{i=1}^{n} \frac{\partial f}{\partial u_i}\frac{\mathrm{d}u_i}{\mathrm{d}x} \tag{B.3}$$

附录 C

向量和矩阵导数

向量、矩阵的导数计算，可以把向量或者矩阵理解为一组标量表达式，对这些表达式分别求导，再把结果组织为向量或者矩阵的形式。本书中组织方式统一采用**分子布局（nominator layout）**。

以下是深度学习反向传播常用到的结论。

向量对向量的导数 此时，自变量是 n 维列向量：

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \tag{C.1}$$

函数 $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$，返回的 m 个实数标量值，可以看成 m 维实数向量 $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$，其中每个 f_i 都返回一个标量结果：

$$\begin{aligned} y_1 &= f_1(\boldsymbol{x}) \\ y_2 &= f_2(\boldsymbol{x}) \\ &\vdots \\ y_m &= f_m(\boldsymbol{x}) \end{aligned}$$

此时，输出向量的函数 $\boldsymbol{f}$ 对 $\boldsymbol{x}$ 的导数，由各个标量值函数 f 相对于向量 $\boldsymbol{x}$ 的梯度组合而成，这个梯度可以继续拆解 $\boldsymbol{x}$ 各维度分量上的偏导数，得到

宽度为 n 的**雅可比 (Jacobian) 矩阵**，形式如下：

$$\frac{\partial \mathbf{y}}{\partial \boldsymbol{x}} = \begin{pmatrix} \nabla f_1(\boldsymbol{x}) \\ \nabla f_2(\boldsymbol{x}) \\ \cdots \\ \nabla f_m(\boldsymbol{x}) \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial \boldsymbol{x}} f_1(\boldsymbol{x}) \\ \frac{\partial}{\partial \boldsymbol{x}} f_2(\boldsymbol{x}) \\ \cdots \\ \frac{\partial}{\partial \boldsymbol{x}} f_m(\boldsymbol{x}) \end{pmatrix} \tag{C.2}$$

$$= \begin{pmatrix} \frac{\partial}{\partial x_1} f_1(\boldsymbol{x}) & \frac{\partial}{\partial x_2} f_1(\boldsymbol{x}) & \cdots & \frac{\partial}{\partial x_n} f_1(\boldsymbol{x}) \\ \frac{\partial}{\partial x_1} f_2(\boldsymbol{x}) & \frac{\partial}{\partial x_2} f_2(\boldsymbol{x}) & \cdots & \frac{\partial}{\partial x_n} f_2(\boldsymbol{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} f_m(\boldsymbol{x}) & \frac{\partial}{\partial x_2} f_m(\boldsymbol{x}) & \cdots & \frac{\partial}{\partial x_n} f_m(\boldsymbol{x}) \end{pmatrix} \tag{C.3}$$

上述过程，由于理解为对向量 $\boldsymbol{x}$ 各个分量求偏导，所以每个 $\frac{\partial}{\partial \boldsymbol{x}} f_i(\boldsymbol{x})$ 都看成是一个 n 维行向量。其中较常见的情况是 $m = n$，函数 $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$ 对 $\boldsymbol{x}$ 各个维度上的分量分别进行特定计算，即 $f_i(\boldsymbol{x}) = f(x_i)$ ，Jacobian 矩阵成为对角方阵：

$$\frac{\partial \mathbf{y}}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial}{\partial \boldsymbol{x}} f_1(\boldsymbol{x}) \\ \frac{\partial}{\partial \boldsymbol{x}} f_2(\boldsymbol{x}) \\ \cdots \\ \frac{\partial}{\partial \boldsymbol{x}} f_m(\boldsymbol{x}) \end{pmatrix} \tag{C.4}$$

$$= \begin{pmatrix} \frac{\partial}{\partial x_1} f_1(\boldsymbol{x}) & \frac{\partial}{\partial x_2} f_1(\boldsymbol{x}) & \cdots & \frac{\partial}{\partial x_n} f_1(\boldsymbol{x}) \\ \frac{\partial}{\partial x_1} f_2(\boldsymbol{x}) & \frac{\partial}{\partial x_2} f_2(\boldsymbol{x}) & \cdots & \frac{\partial}{\partial x_n} f_2(\boldsymbol{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} f_m(\boldsymbol{x}) & \frac{\partial}{\partial x_2} f_m(\boldsymbol{x}) & \cdots & \frac{\partial}{\partial x_n} f_m(\boldsymbol{x}) \end{pmatrix} \tag{C.5}$$

$$= \begin{pmatrix} \frac{\partial}{\partial x_1} f(x_1) & \frac{\partial}{\partial x_2} f(x_1) & \cdots & \frac{\partial}{\partial x_n} f(x_1) \\ \frac{\partial}{\partial x_1} f(x_2) & \frac{\partial}{\partial x_2} f(x_2) & \cdots & \frac{\partial}{\partial x_n} f(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} f(x_n) & \frac{\partial}{\partial x_2} f(x_n) & \cdots & \frac{\partial}{\partial x_n} f(x_n) \end{pmatrix} \qquad (\ j \neq i \ \ \frac{\partial}{\partial x_j} x_i = 0) \tag{C.6}$$

$$= \begin{pmatrix} \frac{\partial}{\partial x_1} f(x_1) & 0 & \cdots & 0 \\ 0 & \frac{\partial}{\partial x_2} f(x_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\partial}{\partial x_n} f(x_n) \end{pmatrix} \quad \blacksquare \tag{C.7}$$

f 函数标量结果构成的对角阵，也记为

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \mathrm{diag}[\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}] = \mathrm{diag}[f'(\boldsymbol{x})] \qquad \text{（结论 1）} \tag{C.8}$$

结论 1，常用于神经网络模型中，各类函数激活环节的反向传播计算。

向量求导的链式法则 假设函数 $\boldsymbol{g}: \mathbb{R} \to \mathbb{R}^k$, 自变量为实数标量 x ，输出 k 维实数向量 $\boldsymbol{u}$，若函数 $\boldsymbol{f}: \mathbb{R}^k \to \mathbb{R}^m$ 以向量 $\boldsymbol{u}$ 作为输入, 输出 m 维实数向量 $\boldsymbol{y}$, 即 $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{g}(x))$，则 $\boldsymbol{y}$ 相对于标量 x 的导数为

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}x} = \begin{pmatrix} \frac{\mathrm{d}f_1(\mathbf{g})}{\mathrm{d}x} \\ \frac{\mathrm{d}f_2(\mathbf{g})}{\mathrm{d}x} \\ \vdots \\ \frac{\mathrm{d}f_m(\mathbf{g})}{\mathrm{d}x} \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1}{\partial g_1}\frac{\mathrm{d}g_1}{\mathrm{d}x} + \frac{\partial f_1}{\partial g_2}\frac{\mathrm{d}g_2}{\mathrm{d}x} + \cdots + \frac{\partial f_1}{\partial g_k}\frac{\mathrm{d}g_k}{\mathrm{d}x} \\ \frac{\partial f_2}{\partial g_1}\frac{\mathrm{d}g_1}{\mathrm{d}x} + \frac{\partial f_2}{\partial g_2}\frac{\mathrm{d}g_2}{\mathrm{d}x} + \cdots + \frac{\partial f_2}{\partial g_k}\frac{\mathrm{d}g_k}{\mathrm{d}x} \\ \vdots \\ \frac{\partial f_m}{\partial g_1}\frac{\mathrm{d}g_1}{\mathrm{d}x} + \frac{\partial f_m}{\partial g_2}\frac{\mathrm{d}g_2}{\mathrm{d}x} + \cdots + \frac{\partial f_m}{\partial g_k}\frac{\mathrm{d}g_k}{\mathrm{d}x} \end{pmatrix} \tag{C.9}$$

上式右边的每一行是分量 f_i 对 $\boldsymbol{g}$ 的梯度向量，它和微分向量 $\mathrm{d}\boldsymbol{x}$ 的相乘，得到全微分 $\mathrm{d}f_i$。

观察 $\frac{\partial f_i}{\mathrm{d}g_j}\frac{\mathrm{d}g_i}{\mathrm{d}x}$ 项，如果把 $\frac{\mathrm{d}g_i}{\mathrm{d}x}$ 拆出来成为向量，这个结果可以看成如下矩阵和向量相乘：

$$\begin{pmatrix} \frac{\partial f_1}{\partial g_1} & \frac{\partial f_1}{\partial g_2} & \cdots & \frac{\partial f_1}{\partial g_k} \\ \frac{\partial f_2}{\partial g_1} & \frac{\partial f_2}{\partial g_2} & \cdots & \frac{\partial f_2}{\partial g_k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial g_1} & \frac{\partial f_m}{\partial g_2} & \cdots & \frac{\partial f_m}{\partial g_k} \end{pmatrix} \begin{pmatrix} \frac{\mathrm{d}g_1}{\mathrm{d}x} \\ \frac{\mathrm{d}g_2}{\mathrm{d}x} \\ \vdots \\ \frac{\mathrm{d}g_k}{\mathrm{d}x} \end{pmatrix} = \frac{\partial \mathbf{f}}{\partial \mathbf{g}}\frac{\mathrm{d}\mathbf{g}}{\mathrm{d}x} \tag{C.10}$$

这个结果和标量求导的链式法则有同样的形式。不同的是，根据矩阵和向量乘法的定义，上式右边两项不满足交换律。

在函数为 $\boldsymbol{g}: \mathbb{R}^n \to \mathbb{R}^k$ 时，向量 $\boldsymbol{x}$ 为自变量的情况下，这个规则同样适

用：

$$\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}(\boldsymbol{g}(\boldsymbol{x})) = \begin{pmatrix} \frac{\partial f_1}{\partial g_1} & \frac{\partial f_1}{\partial g_2} & \cdots & \frac{\partial f_1}{\partial g_k} \\ \frac{\partial f_2}{\partial g_1} & \frac{\partial f_2}{\partial g_2} & \cdots & \frac{\partial f_2}{\partial g_k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial g_1} & \frac{\partial f_m}{\partial g_2} & \cdots & \frac{\partial f_m}{\partial g_k} \end{pmatrix} \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \cdots & \frac{\partial g_1}{\partial x_n} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \cdots & \frac{\partial g_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_k}{\partial x_1} & \frac{\partial g_k}{\partial x_2} & \cdots & \frac{\partial g_k}{\partial x_n} \end{pmatrix} \tag{C.11}$$

$$= \frac{\partial \mathbf{f}}{\partial \mathbf{g}} \frac{\partial \mathbf{g}}{\partial \boldsymbol{x}} \quad \text{（结论 2）} \tag{C.12}$$

同样可以证明，多变量函数对多自变量可微分时，全导数规则在向量和矩阵的导数计算场景下，仍然适用。

结论 2，常用于多层模型误差传递时，化简反向传播表达式，实践中可减少内存占用。

向量按位计算的导数 若有两个同维度向量 $\boldsymbol{w}$ 和 $\boldsymbol{x}$ 的逐元素计算：

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{w}) \bigcirc \boldsymbol{g}(\boldsymbol{x})$$

其中 $\bigcirc$ 是逐位加、减、乘、除 $\oplus \ominus \otimes \oslash$ 中的任意一种，计算表达式可以展开为

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} f_1(\boldsymbol{w}) \bigcirc g_1(\boldsymbol{x}) \\ f_2(\boldsymbol{w}) \bigcirc g_2(\boldsymbol{x}) \\ \vdots \\ f_n(\boldsymbol{w}) \bigcirc g_n(\boldsymbol{x}) \end{pmatrix} \tag{C.13}$$

$\boldsymbol{y}$ 对自变量 $\boldsymbol{x}$ 的导数为

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial}{\partial x_1}(f_1(\boldsymbol{w}) \bigcirc g_1(\boldsymbol{x})) & \frac{\partial}{\partial x_2}(f_1(\boldsymbol{w}) \bigcirc g_1(\boldsymbol{x})) & \cdots & \frac{\partial}{\partial x_n}(f_1(\boldsymbol{w}) \bigcirc g_1(\boldsymbol{x})) \\ \frac{\partial}{\partial x_1}(f_2(\boldsymbol{w}) \bigcirc g_2(\boldsymbol{x})) & \frac{\partial}{\partial x_2}(f_2(\boldsymbol{w}) \bigcirc g_2(\boldsymbol{x})) & \cdots & \frac{\partial}{\partial x_n}(f_2(\boldsymbol{w}) \bigcirc g_2(\boldsymbol{x})) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1}(f_n(\boldsymbol{w}) \bigcirc g_n(\boldsymbol{x})) & \frac{\partial}{\partial x_2}(f_n(\boldsymbol{w}) \bigcirc g_n(\boldsymbol{x})) & \cdots & \frac{\partial}{\partial x_n}(f_n(\boldsymbol{w}) \bigcirc g_n(\boldsymbol{x})) \end{pmatrix} \tag{C.14}$$

对于逐元素运算，f_i, g_i 分别是对应 w_i, x_i 的函数，所以当 $j \neq i$ 时，

$$\frac{\partial}{\partial x_j}(f_i(\boldsymbol{w}) \bigcirc g_i(\boldsymbol{x})) = 0 \tag{C.15}$$

结果是一个对角方阵：

$$\frac{\partial \mathbf{y}}{\partial \boldsymbol{x}} = \text{diag}\left[\frac{\partial}{\partial x_1}(f_1(w_1) \bigcirc g_1(x_1)), \ldots, \frac{\partial}{\partial x_n}(f_n(w_n) \bigcirc g_n(x_n))\right] \tag{C.16}$$

在较常见的场景里：$f_i(w_i) = w_i$，$f_i(x_i) = x_i$，此时：

$$\begin{aligned}
&\frac{\partial(\boldsymbol{w} \oplus \boldsymbol{x})}{\partial \boldsymbol{x}} = \text{diag}(\vec{\mathbf{1}}) = I \\
&\frac{\partial(\boldsymbol{w} \ominus \boldsymbol{x})}{\partial \boldsymbol{x}} = \text{diag}(-\vec{\mathbf{1}}) = -\ I \\
&\frac{\partial(\boldsymbol{w} \otimes \boldsymbol{x})}{\partial \boldsymbol{x}} = \text{diag}(\boldsymbol{w}) \\
&\frac{\partial(\boldsymbol{w} \oslash \boldsymbol{x})}{\partial \boldsymbol{x}} = \text{diag}\left(\cdots \frac{-w_i}{x_i^2} \cdots\right)
\end{aligned} \qquad \text{（结论 3）} \tag{C.17}$$

同理可以推导出上述四种运算下的 $\dfrac{\partial \boldsymbol{y}}{\partial \boldsymbol{w}}$。

结论 3，常用于 RNN 模型各种变体算法中，逐元素计算节点的反向传播推导。

向量函数求和、偏导数的表达式化简 若函数 $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^n$, 对向量 $\boldsymbol{x}$ 各个维度上的分量，分别特定计算，即 $f_i(\boldsymbol{x}) = f(x_i)$；对函数结果求和，注意不是对自变量求和，得到标量 y：

$$y = \text{sum}(\boldsymbol{f}(\boldsymbol{x})) = \sum_{i=1}^{n} f_i(\boldsymbol{x}) \tag{C.18}$$

则函数求和结果对向量 $\boldsymbol{x}$ 的导数为

$$\frac{\partial y}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial y}{\partial x_1} & \frac{\partial y}{\partial x_2} & \cdots & \frac{\partial y}{\partial x_n} \end{pmatrix} \tag{C.19}$$

$$= \begin{pmatrix} \frac{\partial}{\partial x_1}\sum_i f_i(\boldsymbol{x}) & \frac{\partial}{\partial x_2}\sum_i f_i(\boldsymbol{x}) & \cdots & \frac{\partial}{\partial x_n}\sum_i f_i(\boldsymbol{x}) \end{pmatrix} \tag{C.20}$$

$$= \begin{pmatrix} \sum_i \frac{\partial f_i(\boldsymbol{x})}{\partial x_1} & \sum_i \frac{\partial f_i(\boldsymbol{x})}{\partial x_2} & \cdots & \sum_i \frac{\partial f_i(\boldsymbol{x})}{\partial x_n} \end{pmatrix} \tag{C.21}$$

若 $f_i(\boldsymbol{x}) = x_i$, 即 $y = \mathrm{sum}(\boldsymbol{x})$, 上式成为

$$\nabla_{\boldsymbol{x}} y = \begin{pmatrix} \sum_i \frac{\partial f_i(\boldsymbol{x})}{\partial x_1} & \sum_i \frac{\partial f_i(\boldsymbol{x})}{\partial x_2} & \cdots & \sum_i \frac{\partial f_i(\boldsymbol{x})}{\partial x_n} \end{pmatrix} \tag{C.22}$$

$$= \begin{pmatrix} \sum_i \frac{\partial x_i}{\partial x_1} & \sum_i \frac{\partial x_i}{\partial x_2} & \cdots & \sum_i \frac{\partial x_i}{\partial x_n} \end{pmatrix} \quad \text{（结论 4）} \tag{C.23}$$

由于 $j \neq i$ 时，$\frac{\partial}{\partial x_j} x_i = 0$，所以上式可以进一步简化为

$$\begin{aligned} \nabla_{\boldsymbol{x}} y &= \begin{pmatrix} \sum_i \frac{\partial x_i}{\partial x_1} & \sum_i \frac{\partial x_i}{\partial x_2} & \cdots & \sum_i \frac{x_i}{\partial x_n} \end{pmatrix} \\ &= \begin{pmatrix} \frac{\partial x_1}{\partial x_1} & \frac{\partial x_2}{\partial x_2} & \cdots & \frac{\partial x_n}{\partial x_n} \end{pmatrix} = \vec{\mathbf{1}}^{\mathrm{T}} \end{aligned} \tag{C.24}$$

结论 4，常用在向量内积运算的反向传播环节，因为向量的内积运算就是对应元素乘积项的求和。

仿射变换求导 由向量内积的定义：$\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \boldsymbol{x}^{\mathrm{T}} \boldsymbol{y}$

函数 $y = \boldsymbol{w} \cdot \boldsymbol{x} + b$，权值向量和自变量的内积运算，表达为 $\sum_i^n (w_i x_i)$，可分解为逐元素乘积与求和两步运算。

为原函数引入向量 $\boldsymbol{u}$ 作为中间变量：

$$\boldsymbol{u} = \boldsymbol{w} \otimes \boldsymbol{x} \tag{C.25}$$

$$y = \mathrm{sum}(\boldsymbol{u}) \tag{C.26}$$

就可以运用已证明的**结论 2** 向量求导链式法则、**结论 3** 向量的逐元素运算和**结论 4** 向量函数求和偏导数约简的方法，进一步计算导数：

$$\begin{aligned} \frac{\partial y}{\partial \boldsymbol{w}} &= \frac{\partial y}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \boldsymbol{w}} = \vec{\mathbf{1}}^{\mathrm{T}} \cdot \mathrm{diag}(\boldsymbol{x}) = \boldsymbol{x}^{\mathrm{T}} \\ \frac{\partial y}{\partial \boldsymbol{x}} &= \frac{\partial y}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \boldsymbol{x}} = \vec{\mathbf{1}}^{\mathrm{T}} \cdot \mathrm{diag}(\boldsymbol{w}) = \boldsymbol{w}^{\mathrm{T}} \end{aligned} \quad \text{（结论 5）} \tag{C.27}$$

结论 5，可用于仿射变换环节的误差传递与权参梯度计算，在深度学习各类核心支撑模型的反向传播算法中，是最常用到的结论。

附录 D

概率论和数理统计

期望　随机变量的均值，记作 $\mathbb{E}(X)$。

对于离散型随机变量 X，它是随机变量各个取值 x 和对应概率 p 的乘积之和。

$$\mathbb{E}(X) = \sum_{i=1}^{\infty} x_i p_i \tag{D.1}$$

方差　用于度量随机变量 X 与它的期望 $\mathbb{E}(X)$ 的偏离程度，记作 $\mathrm{Var}(X)$。

对于离散型随机变量，它是随机变量各个取值与期望之差的平方与对应概率的乘积之和。

$$\mathrm{Var}(X) = \sum_{i=1}^{\infty} [x_i - \mathbb{E}(X)]^2 p_i \tag{D.2}$$

常用分布

0-1 分布　设离散型随机变量 X 只能取 0 与 1 两个值，它的分布律是

$$P\{X = k\} = p^k (1-p)^{1-k} \qquad \text{其中} \quad k = 0, 1 \quad \text{且} \quad 0 < p < 1$$

则称 X 服从以 p 为参数的 0-1 分布。

二项分布　设实验 E 只有两种可能结果，A 及 $\overline{A}$，则 E 称为伯努利试验；若将试验 E 独立地重复进行 n 次，则称这个试验为 n 重伯努利实验。

在 n 重伯努利实验中，若 $P(A) = p,\ P(\overline{A}) = 1-p.$ 记 X 为 n 次试验中事件 A 发生的次数，显然 X 是一个离散型随机变量，它的取值为 $0, 1, 2, \cdots, n.$

它的分布率为

$$P\{X=k\}=C_n^k p^k(1-p)^{n-k} \qquad \text{其中} \quad k=0,1,2,\cdots,n$$

称 X 服从参数为 n,p 的二项分布，记为

$$X\sim B(n,p)$$

若 $n=1$，二项分布就成了 0-1 分布。

正态分布 若连续型随机变量 X 的概率密度函数为

$$f(x)=\frac{1}{\sqrt{2\pi}\sigma}\mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}} \ (-\infty<x<+\infty) \tag{D.3}$$

其中 μ 与 $\sigma>0$ 都是常数，则称 X 服从参数为 μ 和 σ 的正态分布，记为

$$X\sim N(\mu,\sigma^2)$$

当 $\mu=0$，$\sigma^2=1$ 时，称随机变量 X 服从标准正态分布。

抽样样本的均值和方差 设 $X_1,X_2,\cdots,X_n$ 是随机变量 X 的样本。

样本均值为

$$\overline{X}=\frac{1}{n}\sum_{i=1}^{n}X_i \tag{D.4}$$

样本方差为

$$s^2=\frac{1}{n-1}\sum_{i=1}^{n}(X_i-\overline{X})^2 \tag{D.5}$$

大数定理 独立同分布的随机变量 $X_1,X_2,\cdots,X_n$，数学期望为 $\mathbb{E}(X)=\mu$ 存在且为有限值, 当 n 充分大时，算术均值 $\overline{X}=\dfrac{1}{n}\sum\limits_{i=1}^{n}X_i$，很可能接近 μ。

中心极限定理 独立同分布的随机变量 $X_1, X_2, \cdots, X_n$，数学期望为 $\mathbb{E}(X) = \mu$，方差 $\mathrm{Var}(X) = \sigma^2 > 0$，当 n 充分大时，算术平均值 $\overline{X} = \dfrac{1}{n}\sum_{i=1}^{n} X_i$ 的分布近似地服从均值为 μ，方差为 $\dfrac{\sigma^2}{n}$ 的正态分布。

附录部分包含了书中讨论的深度学习算法所涉及的数学基础，方便初学者速查和理解其直观意义。如需概念的严格定义和展开论证，可参考相关教材和专著。

参 考 文 献

[1] The Stanford CS class CS229. `http://cs229.stanford.edu/`

[2] Pang-Ning Tan, Michael Steinbach, Vipin Kumar. **Introduction to Data Mining**. Pearson 2005 （中译本 **数据挖掘导论**. 范明，范宏建等译. 北京：人民邮电出版社，2011）

[3] Terence Parr, Jeremy Howard. **The Matrix Calculus You Need For Deep Learning**, 2018.

后记

正确使用模型和各种深度学习框架，离不开对原理的了解，如果对**整体原理**了然于胸，在应用深度学习框架的时候，可以避免陷入“盲人摸象”的窘境，看清全貌，直达本质，解决工程实践中遇到的问题。

Talk is cheap, show me the model.

实践是对理解进行校验和纠偏的最佳途径。有侧重地理解**核心和基础的算法**，实现之，有助于复现业内新发布的方法。如果结合自身场景，能够优化改进现有算法，还可以扩展已知方法的边界。

谨此，共勉。

索引